Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science and Engineering

Real-time Skeletal Animation

by

Ladislav Kavan

A doctoral thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Ph.D. Programme: Electrical Engineering and Information Technology
Branch of study: Computer Science and Engineering

June 2007

Thesis Supervisor:
      Doc. Ing. Jiří Žára, Csc.
      Department of Computer Science and Engineering
      Faculty of Electrical Engineering
      Czech Technical University in Prague
      Karlovo náměstí 13
      121 35 Prague 2
      Czech Republic

Abstract and Contributions

Skeletal animation is a popular technique in real-time virtual reality applications. It is used to obtain believable yet efficiently computable deformations of 3D objects, such as virtual humans or animals. This thesis aims to study skeletal animation from both practical and theoretical viewpoints. The goal is to provide foundations for next-generation skeletal animation sub-systems, offering more realistic deformations, direct support of collision detection and compression of arbitrary animations.

Specifically, the contributions of this thesis consist of:

1. Spherical blend skinning [A.6], a new skin deformation algorithm delivering more realistic skin deformation while retaining run-time performance comparable to the classical method (linear blend skinning).

2. Study of rigid transformation blending algorithms and comparison of their mathematical properties [A.8, A.2], with the goal of finding an optimal blending algorithm for skinning. We demonstrate how such a blending algorithm can be derived using dual quaternions.

3. Collision detection algorithm for models deformed by linear blend skinning [A.1]. The presented algorithm is able to achieve a sublinear time complexity (with respect to the number of vertices), outperforming previous collision detection methods significantly.

4. Collision detection for spherical blend skinning [A.4], generalizing the previous algorithm from linear to spherical blending. Even though the case of spherical blending is more complex, the resulting algorithm is again sublinear and almost as fast as the previous linear version.

5. An algorithm to automatically construct skinned approximations of an arbitrary animation [A.3, A.9]. This enables to exploit the benefits of skinned animations (i.e., data reduction and efficient rendering) for a broader class of 3D objects, including, e.g., cloth.

Keywords:

skeletal animation, skinning, collision detection, linear blending, spherical blending, dual quaternions, animation compression, skinning approximation

Acknowledgements

Dedication

To my parents.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

This thesis studies real-time animation of skeletally deformable objects, which is an important topic in computer animation and virtual reality. We present new algorithms for realistic skin deformation, fast and accurate collision detection and automatic authoring of skeletal animations. While the proposed algorithms are designed for practical use, we pay an equal attention to the related theoretical background.

1.1 Motivation

In the past decades, the research in 3D computer graphics focused mainly on rendering of high-fidelity images. Less attention has been paid to the problems related to motion and simulation. As a result, contemporary visualization methods perform very well in rendering still images. Overstating only a little bit, some experts claim even that "In rendering still images, we are done" [21]. This indicates that a new research challenge in computer graphics is to obtain this level of realism also for *animation*.

A description of motion is relatively simple for rigid objects, i.e., those having constant, unchanging shape. In this case, any instantaneous motion is a *screw* motion [68], which is fairly well studied, especially thanks to the field of robotics. However, objects in the real world are seldom rigid. An everyday example of motion is the motion of human beings or animals, which is not rigid at all – not even approximately. Obviously, motion of organic entities is far more complex than motion of a rigid object.

It would be of course possible to focus on one specific class of objects, such as human figures, and study their animation in detail. This would enable us to exploit an a priori knowledge, e.g., of the human anatomy. On the other hand, this approach is too restrictive and in this thesis, we therefore follow a different way. We focus on a specific *deformation model*, instead of a specific class of objects from the real world. Generally, deformation model is a mathematical description of how the model's shape changes with respect to a given set of parameters (controls). The deformation model we study in this thesis is known as skeletal animation (or skinning, skeletal subspace deformation, matrix palette skinning or simply enveloping). This model has been originally proposed for the animation of virtual humanoids, but practice has shown its much wider applicability. Skeletal deformation can be advantageously used to animate also various kinds of animals (not only vertebrates, as the name could suggest), plants, cloth etc.

The importance of skeletal deformation is not only in the fact that it accommodates a wide range of objects from the real world. Another big motivation is its industrial importance – many modern 3D computer games use skeletal deformation to animate avatars and other inhabitants of virtual worlds. As a result, commodity graphics hardware is designed in order to efficiently support skeletal animation [58].

The computer games industry would itself be a compelling justification to investigate skeletal animation. However, videogames are not the only possible application: the same model can be used, e.g., in architectonical studies for crowd simulation, distributed virtual environments or emergency training [92]. Apart from its importance in practice, skinning raises also interesting theoretical questions, which have not been explicitly addressed before.

1.2   Realistic Skin Deformation

A deformation model, such as skeletal animation, is an abstract description of how the shape of a 3D object changes with respect to a small number of control parameters. There does not exist only one skeletal deformation technique: skeletal animation embraces a whole class of related methods. However, the basic structure of a skeletally controlled model is always the same. The object to be deformed is specified by its boundary representation, i.e., a triangular mesh with an arbitrary connectivity. Alternative representations are also possible, such as volumetric or point-based, but the boundary representation is by far the most common one.

In general, skeletal animation could theoretically refer to any technique that computes shape of the skin for a given skeletal posture. However, the term *skeletal animation* is usually used only when there exists a direct geometric relationship between the skeleton and skin – in this thesis, we will also adhere to this meaning. In our case, the deformation structure, i.e., the parameters controlling the deformation, is simply a list of 3D transformations (often represented by matrices, which explains the synonym *matrix palette skinning*). Traditionally, these transformations describe the actual position and orientation of the joints in the animated skeleton. However, as has been pointed out recently [63], skeleton is actually not necessary for skinning – the transformations alone are sufficient.

Typically, only the joints important to achieve the desired animation effects are modelled, see Figure 1.1 (note for example the simplification of the spine). Of course, such simple control structures cannot be expected to model deformations of human skin accurately. However, this is not the goal of skeletal animation: its goal is to provide a reasonable approximation with minimal computational complexity. This is what is often required in practical real-time applications.

Obviously, there is a trade-off between visual quality of the deformed skin and run-time efficiency of the algorithm. This thesis focuses on real-time animation, i.e., on algorithms which are able to deliver results within tens of milliseconds on a commodity hardware. With this time budget, it is quite challenging to create a realistic deformation – especially for the human body, where everyone immediately notices even small artifacts. Nevertheless, the constraints of real-time applications are strict. In fact, artifacts in the skin deformation are more acceptable than a slow-down of the rendering process, which results in lagging – an even more disturbing shortcoming. This explains why the skeletal animation algorithms used in the industrial practice indeed produce artifacts at certain circumstances.
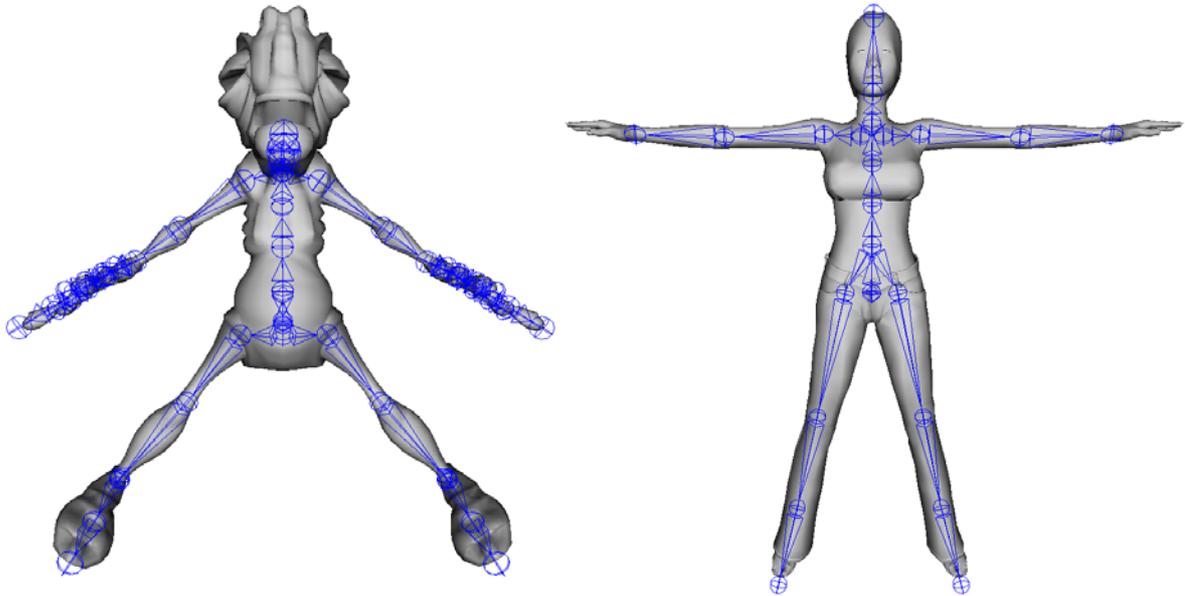
Figure 1.1: Typical skeletally deformable objects.

However, this raises a question: are such compromises really necessary because of the very nature of the problem? Specifically, we might ask whether it is not possible to design a skin deformation algorithm with reasonable time complexity, but avoiding most of the artifacts.

Our first alternative skinning algorithm, spherical blend skinning, is presented in Chapter 3. This algorithm resolves most artifacts of the previous standard solution (linear blend skinning). This is achieved at small overhead in computational complexity and no overhead at all in model creation (our method works with the same input data as linear blend skinning). Unfortunately, this algorithm is not easy to implement because it involves a complicated algorithm as a subroutine (singular value decomposition). Also, the skin deformation produced by spherical skinning is still not 100% artifact free.

Therefore, in Chapter 4, we isolate the geometrical problems arising in skinning and formulate the properties of an optimal blending algorithm for skinning. We review previous rigid transformation blending methods and compare their properties, concluding that no previous algorithm is optimal for skinning. Therefore, we propose new methods based on dual quaternions, which enable us to obtain an artifact free skinning with a very simple and efficient GPU implementation.

1.3   Collision Detection

Realistic skin deformation, discussed in the previous section, is not the only problem with simulation of inhabited virtual environments. Skeletally deformable models typically in-

teract with other models, either rigid or also deformable ones. The first step in handling these interactions is to detect geometric interpenetrations (or collisions, see Figure 1.2). In the case of skeletally deformable objects, collision detection is strongly influenced by the chosen skin deformation algorithm.
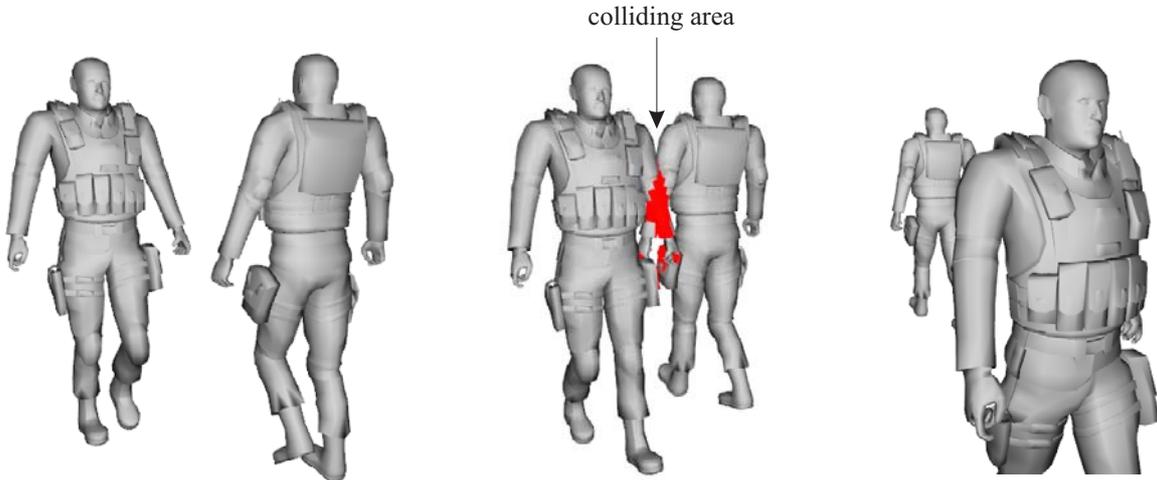
Figure 1.2: A collision of two virtual characters.

The strict time budget of real-time applications applies to this problem as well. It would be of course possible to neglect the fact that the objects are deformed by an underlying skeleton, and apply a general collision detection algorithm. However, this is not the most efficient approach. If we exploit the knowledge about the actual deformation algorithm, we can obtain faster collision detection.

In practice, the high computational complexity of collision detection is typically resolved by considering only a rough approximation of the true shape, i.e., working only with a substitute collision object. This substitute geometry is usually very simple and thus allows very efficient collision detection queries. However, this of course introduces artifacts: some interpenetrations can be missed or the objects can be reported to collide before they actually do.

We can ask ourselves whether this approximation – even though being a common practice – is really necessary. One possible way to achieve exact but efficient collision detection is to focus on a specific deformation model and exploit its special properties [81, 62]. We pursue this idea first for the case of linear blend skinning (Chapter 5), subsequently generalizing to spherical blend skinning (Chapter 6). In both cases, we obtain a significant speedup over general collision detection methods, while still supporting a very rich class of 3D objects.

1.4 Authoring Skeletal Animations

In recent years, 3D animators have demonstrated that skeletal animation can be used not only for virtual humanoids, but also for various other kinds of creatures, animals, plants and even cloth – see Figure 1.3. However, the effort the animators invest into the creation of such skeletally deformable models is remarkable – authoring of one skeletal model requires hours or even days of labor (this can, however, be ameliorated using more advanced tools, such as those presented in [103]). The process is more intuitive for objects which naturally possess skeleton, such as human or animal figures. For other objects, however, the animators must use a lot of imagination to place the joints in appropriate locations (in fact anticipating the most probable prospective deformations).

Figure 1.3: Lord of the Rings videogame (©2004 Electronic Arts). Skeletal animation is used to animate also the wizard's cloak, besides the character itself.

Of course, it is more convenient to animate the objects made of cloth or highly elastic materials by other techniques than skeletal animation. For the case of cloth, specialized tools are available in professional software [9, 10]. Animations created with this software are very realistic, but when exported, they consume a lot of memory and are thus hard to manipulate with. This is one of the reasons why animators are concerned with creation of skinned approximations. Other reason is that practically all 3D engines natively support skeletal animation (unlike other data reduction methods).

It would be advantageous to have a tool for an automatic conversion from general deformable animation (such as the output from a cloth simulator) to a skeletal one. This would remove the burden of constructing skeletal approximations by hand. Such a tool has been presented by James and Twigg [63] for the class of quasi-articulated objects (e.g., those of human or animal figures).

Our algorithm is based on a different principle and is able to construct efficient skinned approximations of arbitrary deformations, including highly deformable ones, such as those of cloth or elastic materials, see Figure 1.4.
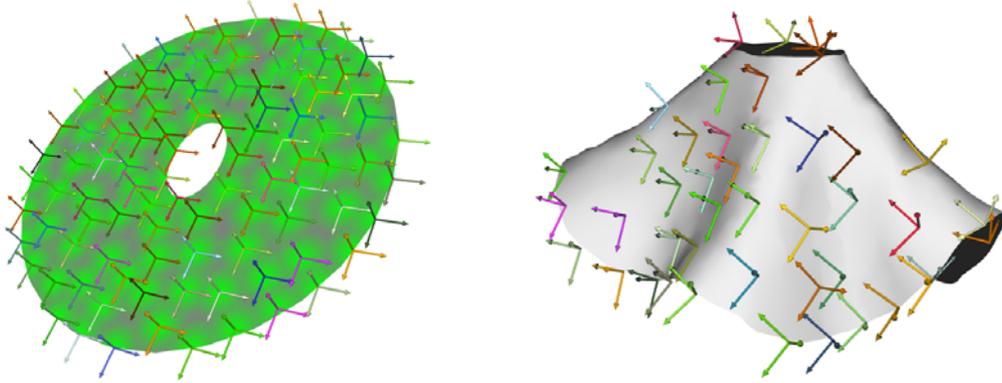
Figure 1.4: A skirt model deformed by skeletal animation (left: reference pose, right: deformed one). The skinning is computed automatically by our method.

1.5   Related Work

This section presents a brief summary of previous work, focusing on the methods most closely related to this thesis. Please refer to Chapter 2 for a more detailed background, covering also other important yet not so closely related approaches.

1.5.1   Real-time Skin Deformations

The idea of controlling the shape of a model using an underlying set of joints was pioneered by Magnenat-Thalmann et al. [90]. The first skeletal animation algorithms used in real-time applications were very simple, such as the most popular *linear blend skinning* [76, 77]. The artifacts of linear blend skinning were discovered soon in the game development community [134]. A trick was suggested how to alleviate the artifacts, based on introduction of auxiliary bones. At the same time, acceleration of linear blend skinning by programmable graphics hardware was proposed [87] (without any treatment of the artifacts).

A natural way to combat the artifacts is by using more example skins, not just one. This is the idea of example based methods [85, 120]. However, the amount of memory necessary to store the example meshes can be an issue. Memory consumption can be improved by applying principal component analysis, as shown in [74]. Another example-based method attacking the linear blend skinning artifacts was developed by researchers from Industrial Light and Magic [132]. Their method is based on a more general blending scheme, with parameters optimized in order to minimize the least squares distance from example meshes. The example based method presented in [99] solves the artifacts by adding auxiliary joints and automatically derives blending weights from example meshes. This enables also the simulation of muscle bulging effects within the framework of skeletal animation.

The example based methods are able to produce a very realistic skin deformation – limited only by the number of example skins. Unfortunately, the authoring of example skins is costly, because each of them must be itself a realistic 3D model. Therefore, cheaper ways of fixing the skinning artifacts were sought. A useful tool is presented in [100]. Even though it does not solve the artifacts, it simplifies the design of skeletally animated models and visualizes the range of possible deformations. An interesting technique to overcome the linear blend skinning artifacts is presented in [89, 28]. This method is based on a more sophisticated matrix blending algorithm [3]. The big advantage of this approach is that it does not require any extra data or example skins – it works with the same input as linear blend skinning. However, the drawback is in the higher computational complexity, stemming from the advanced matrix blending method [3]. An algorithm with similar features, but based on a different concept was introduced in [56]. This solution is based on quaternion blending, which is more computationally efficient. Unfortunately, the method presented in [56] is not applicable to all skeletally deformable objects, but only to those satisfying a simplifying assumption.

In summary, all alternative geometric skinning algorithms presented so far successfully combat the artifacts of linear blend skinning, but also introduce trade-offs to consider. Therefore, many practical applications still opt for the simple linear blend skinning.

1.5.2 Collision Detection

The problem of collision detection is very well studied for the case of rigid objects [59, 39, 72, 54, 65, 33]. This is because rigid objects lend themselves to pre-processing, exploiting the obvious fact that their shape does not change during simulation. However, this is not the case of deformable objects – collision detection of deformable objects is thus more challenging and remains an active research area. A nice survey of recent deformable collision detection algorithms presents Teschner et al. [124].

One possible approach to deformable collision detection is based on spatial hashing [123]. More literature is devoted to image-space techniques [70, 55, 44, 42, 43]. Image-space methods do not require any pre-processing and therefore are naturally suitable for deformable collision detection. They can also take advantage of the programmable graphics hardware. However, in most cases, the accuracy of image-space techniques is limited by the resolution of the projection plane.

Another approach to deformable collision detection is to adapt previous established algorithms for rigid objects, especially those based on bounding volume hierarchies. The first articles on this topic consider axis aligned bounding boxes [128, 80] and spheres [20]. These approaches are based on refitting of complete bounding volume hierarchy whenever the object deforms. A more efficient way is to perform refitting in an on-demand way, which however works only for specific deformation models [81, 62]. Such algorithms are very fast, with speed comparable to rigid body collision detection [62]. Another way to speed up deformable collision detection is by exploiting spatial coherency [82].

Special attention has been devoted to the problem of collision detection for moving cloth [130, 107, 129, 97, 28], which is especially challenging due to frequent self-collisions. A rather brute force GPU algorithm has been proposed by Choi at el. [25]. Recently, a more efficient algorithm based on chromatic decomposition has been presented by Govindaraju et al. [41]. Even better results can be achieved by more advanced decomposition methods [40]. The idea of the mesh decomposition is to convert the self-collision detection problem to an easier n-body collision detection.

An important class of collision detection algorithms are so called *time-critical* algorithms, which can be interrupted and asked for an approximate solution. These algorithms are relatively well studied for the case of rigid objects [59, 17, 18]. Recently, one time-critical algorithm has been proposed also for deformable objects [95].

1.5.3   Skinning as Data Reduction

The approximation of general deformable animations by skinning, as discussed in Section 1.4, can be regarded as a method of data reduction. The problem of animation compression has been opened by the pioneering work of Lengyel [84]. A number of more sophisticated algorithms has been proposed subsequently [4, 69, 49, 115]. These methods apply the usual data compression techniques: principal components analysis (global or clustered), linear prediction coding and decorrelation based on wavelets.

An interesting alternative approach to animation compression is to encode the mesh into 2D *geometry images* [47] and apply standard video compression methods [19].

A method resembling the skeletal approximation is construction of a rigid transformation basis [27]. The difference is that no blending of transformations is performed with rigid transformation basis, which sometimes results in non-smooth shapes. Construction of full skeletal approximations has first been discussed in the context of clothed virtual humans [102, 28]. A more general technique appeared recently [63]. This method works by identifying approximately rigid components and creating proxy-bones accordingly. Impressive animation of a huge number of running animals have been presented, but the technique falls short to capture highly deformable models, such as cloth.

1.6   Organization of the Thesis

Chapter 2 discusses the essential background, notation and conventions. It also provides short tutorials on quaternions and dual quaternions – algebras important in the subsequent chapters. Furthermore, a broader summary of previous work on skin deformation and collision detection is provided.

Chapter 3 presents our first contribution to real-time skin deformation: spherical blend skinning. Although it is still not a perfect solution, it is a considerable improvement over previous methods. Even though a better solution is proposed in Chapter 4, spherical blend

skinning still can be useful in certain situations (e.g., when an efficient collision detection is needed).

Chapter 4 discusses the shortcomings of spherical blend skinning and formulates the ideal properties of rigid transformation blending for skinning. With the aid of dual quaternions, new algorithms for rigid transformation blending are developed and applied to skinning.

Chapter 5 tackles the problem of collision detection for models deformed by linear blend skinning. A new collision detection algorithm is described, achieving considerably higher run-time performance than previous methods.

Chapter 6 discusses how the collision detection algorithm from Chapter 5 can be adapted for spherical blend skinning. This is done by considering rotational bounds in the space of unit quaternions. The result is an algorithm for collision detection between models deformed by spherical blend skinning with speed comparable to the algorithm from Chapter 5.

Chapter 7 presents our algorithm to construct skinning approximations of arbitrary animations. In contrast to previous work, our method facilitates also highly deformable animations, such as those of cloth and elastic materials.

Chapter 8 concludes the thesis with a summary of our contributions and suggestions for future work.

The Appendix contains two proofs that are important for the thesis, but too long to be included within the text of the previous chapters. Some symbolic computations involved in these proofs are carried out using Maple (source code listings included).

2 Background and State-of-the-art

The purpose of this chapter is twofold: first, we recapitulate the mathematical and graphical tools that will be important in the following chapters. Second, we present a broader body of related work, discussing also alternative approaches to skinning and collision detection.

2.1   Preliminaries

This section summarizes our notation, conventions and the most frequently used shorthands. The set of integer numbers is denoted as Z, the set of real numbers as R, and the d-dimensional Euclidean space as R^d. If not stated otherwise, we assume R^3. Scalars are written in lower case *italics*, vectors as well as quaternions in lower case **boldface** and matrices in upper case *ITALICS*. Therefore, for example, x_0, x_1, x_2 means components of a vector $\mathbf{x}$, while $\mathbf{x}_0, \mathbf{x}_1, \mathbf{x}_2$ is a list of three vectors. A closed real interval is denoted as $[a, b] = \{x \in R : a \le x \le b\}$ (we do not introduce any special notation for open or semi-open intervals – we will use explicit expressions instead).

Vectors are by default considered column, for example, the formula $M\mathbf{x} = \mathbf{y}$ should be interpreted as

$$
\begin{pmatrix} m_{00} & m_{01} & m_{02} \\ m_{10} & m_{11} & m_{12} \\ m_{20} & m_{21} & m_{22} \end{pmatrix} \begin{pmatrix} x_0 \\ x_1 \\ x_2 \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \end{pmatrix}
$$

Due to this convention, the product MN means that N is applied before M, which is in accordance with the OpenGL library conventions [118].

Coordinate systems in R^3 are assumed to be right-handed, i.e., when viewed from the positive direction of the $\mathbf{z}$-axis, the rotation from the $\mathbf{x}$-axis to the $\mathbf{y}$-axis runs counterclockwise, see Figure 2.1. This corresponds with the common 3D rotation convention: a rotation given by an axis $\mathbf{a} \in R^3$ and an angle α means a counter-clockwise rotation when viewed from the positive direction of axis $\mathbf{a}$ [32].
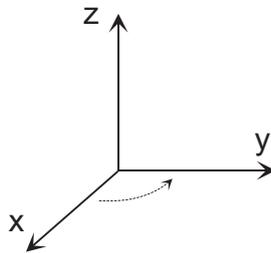
Figure 2.1: Right-handed coordinate system

Any R^3 vector can be represented in *homogeneous coordinates* by an R^4 vector:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \equiv \begin{pmatrix} xh \\ yh \\ zh \\ h \end{pmatrix}$$

where $h \in R \setminus \{0\}$. More information about homogeneous coordinates can be found in [15] or in the classical book [34]. A transformation in homogeneous coordinates can be expressed by a *homogeneous matrix*, which has, by convention, the following structure

$$\begin{pmatrix} m_{00} & m_{01} & m_{02} & t_0 \\ m_{10} & m_{11} & m_{12} & t_1 \\ m_{20} & m_{21} & m_{22} & t_2 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} M & \mathbf{t} \\ 0 & 1 \end{pmatrix}$$

where the matrix M is the original R^3 linear transformation and $\mathbf{t}$ is a translation vector. We do not introduce a different notation for R^3 vectors and their homogeneous R^4 counterparts with the last coordinate equal to 1 (even though this, strictly speaking, should not be considered as homogeneous coordinates, it is the common practice in computer graphics). An analogical convention is used also for matrices. The group of rotations is sometimes called $SO(3)$ and the group of rigid transformations $SE(3)$, as is usual in literature [101].

The identity matrix is denoted as I, and the transpose of a matrix by superscript ^{T}. The standard basis vectors of R^d are denoted as $\mathbf{e}_1 = (1, 0, \ldots, 0)^T, \ldots, \mathbf{e}_d = (0, \ldots, 0, 1)^T$. We denote the dot product of two vectors $\mathbf{v}_1, \mathbf{v}_2$ as $\langle \mathbf{v}_1, \mathbf{v}_2 \rangle$ and the norm $\|\mathbf{v}_1\|$ as a shortcut for $\sqrt{\langle \mathbf{v}_1, \mathbf{v}_1 \rangle}$.

The convex hull of set $A \subseteq R^d$, i.e., the smallest convex set containing A, is denoted as $CH(A)$. In order to simplify the notation of convex combinations, we introduce the set of convex weights

$$W_d = \{\mathbf{x} \in R^d : x_1 \geq 0, \ldots, x_d \geq 0, \ \sum_{i=1}^{d} x_i = 1\}$$

In order to distinguish between single and multi-parameter interpolation, we use the term *blending* for interpolation with more than one independent parameter and reserve the term *interpolation* for single-parametric interpolation (i.e., essentially, for interpolation curves).

2.1.1 Quaternions

This thesis often makes use of quaternion algebra, discovered by W. R. Hamilton [50]. The quaternion representation of 3D rotations is important, because it is singularity-free and has the minimal number of coordinates to achieve this (four). Other popular representations, such as Euler angles, axis-angle or exponential coordinates [46] use only three

coordinates, but do therefore contain a singularity. A matrix representation of rotations is also singularity free, but requires nine coordinates (with six dependencies due to the orthonormality constraints). This is one of the reasons why quaternion representation is so popular in computer graphics. This section contains only basic information on the subject; further details can be found in [52, 29, 32] and especially in the recent book [51]. From a modern point of view, quaternions can also be considered as a special case of more general geometric algebras (also known as Clifford algebras) [94, 57, 133].

A quaternion $\mathbf{q}$ is a four-tuple $\mathbf{q} = w + xi + yj + zk$, where w, x, y, z are real numbers and i, j, k are quaternion units. Number w is sometimes called the *scalar* (or real) part, and x, y, z the *vector* part. The addition and subtraction of quaternions is component-wise, regarding quaternions as R^4 vectors. We assume that $1, i, j, k$ are linearly independent. The multiplication of quaternions is associative and obeys the law $i^2 = j^2 = k^2 = ijk = -1$, from which can be derived the products of all pairs of quaternion units, i.e., $ij = -ji = k, jk = -kj = i, ki = -ik = j$. The product of two general quaternions $\mathbf{q}_0 = w_0 + x_0 i + y_0 j + z_0 k$ and $\mathbf{q}_1 = w_1 + x_1 i + y_1 j + z_1 k$ thus is

$$\begin{aligned}
\mathbf{q}_0 \mathbf{q}_1 \quad = \quad & (w_0 w_1 - x_0 x_1 - y_0 y_1 - z_0 z_1) + \\
& (x_0 w_1 + w_0 x_1 + y_0 z_1 - z_0 y_1)i + \\
& (y_0 w_1 + w_0 y_1 + z_0 x_1 - x_0 z_1)j + \\
& (z_0 w_1 + w_0 z_1 + x_0 y_1 - y_0 x_1)k
\end{aligned}$$

This can be written more concisely, if we treat the vector parts of quaternions $\mathbf{q}_0, \mathbf{q}_1$ as R^3 vectors, $\mathbf{v}_0 = (x_0, y_0, z_0)^T$, $\mathbf{v}_1 = (x_1, y_1, z_1)^T$:

$$\mathbf{q}_0 \mathbf{q}_1 = (w_0 + \mathbf{v}_0)(w_1 + \mathbf{v}_1) = (w_0 w_1 - \langle \mathbf{v}_0, \mathbf{v}_1 \rangle) + w_0 \mathbf{v}_1 + w_1 \mathbf{v}_0 + \mathbf{v}_0 \times \mathbf{v}_1 \qquad (2.1)$$

Quaternion multiplication is associative and distributive, but not commutative (in general). The conjugate of quaternion $\mathbf{q}$ is $\mathbf{q}^* = w - xi - yj - zk$, and the norm of quaternion $\mathbf{q}$ is defined as $\|\mathbf{q}\| = \sqrt{\mathbf{q}^* \mathbf{q}} = \sqrt{\mathbf{q} \mathbf{q}^*} = \sqrt{w^2 + x^2 + y^2 + z^2}$. Conjugation of quaternion product obeys the rule $(\mathbf{p}\mathbf{q})^* = \mathbf{q}^* \mathbf{p}^*$ for any two quaternions $\mathbf{p}, \mathbf{q}$. Unit quaternion is a quaternion with norm 1. Inverse quaternion is defined only if $\mathbf{q} \neq 0$, in which case $\mathbf{q}^{-1} = \frac{\mathbf{q}^*}{\|\mathbf{q}\|^2}$.

A 3D rotation given by unit axis $\mathbf{a} = (a_0, a_1, a_2)^T$ and angle of rotation α can be represented by unit quaternion

$$\mathbf{q} = \cos\left(\frac{\alpha}{2}\right) + \sin\left(\frac{\alpha}{2}\right)(a_0 i + a_1 j + a_2 k)$$

Unit quaternions form a sub-group of quaternions. Number 1, viewed as a quaternion corresponds to the identity rotation (we prefer not to differ scalar and quaternion 1 by the boldface convention, because in this case the difference is rather moot). Conjugation of a unit quaternion yields the inverse rotation. Geometrically, unit quaternions form surface of a hypersphere $S_3 = \{\mathbf{q} \in R^4 : \|\mathbf{q}\| = 1\}$. The correspondence among 3D rotations and unit quaternions is one to two, because both unit quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same

rotation. This fact is sometimes referred as quaternion *antipodality*. This is not a specific feature or drawback of quaternion algebra; actually, this reflects the interesting fact that a 360 degrees rotation is not equivalent with identity, whereas a 720 degrees rotation is (in fact, all rotations about $360 + 720k$ ($k \in Z$) degrees form one group and all $720k$ degrees rotations form the other one). This can be demonstrated even in the real world by the interesting Dirac's belt trick [52, 51].

A 3D vector $\mathbf{v} = (v_0, v_1, v_2)^T$ can be expressed as quaternion $v_0 i + v_1 j + v_2 k$. It is convenient to denote this quaternion by the same symbol as the 3D vector, i.e., $\mathbf{v}$. Using this convention, a rotation of vector $\mathbf{v}$ by unit quaternion $\mathbf{q}$ is given as $\mathbf{v}' = \mathbf{q}\mathbf{v}\mathbf{q}^*$. It can be shown that the scalar part of $\mathbf{v}'$ is zero and the vector part corresponds to rotation of $\mathbf{v}$ about axis $\mathbf{a}$ and angle α. Composition of rotations corresponds to quaternion multiplication, because, if $\mathbf{p}$ is another unit quaternion used to rotate $\mathbf{v}'$, we obtain $\mathbf{v}'' = \mathbf{p}\mathbf{v}'\mathbf{p}^* = \mathbf{p}\mathbf{q}\mathbf{v}\mathbf{q}^*\mathbf{p}^* = \mathbf{p}\mathbf{q}\mathbf{v}(\mathbf{p}\mathbf{q})^*$. From this formula we also see that the quaternion $\mathbf{p}\mathbf{q}$ first performs the rotation $\mathbf{q}$ and second $\mathbf{p}$. This is in accordance with our matrix multiplication convention (Section 2.1).

The Euler's identity for complex numbers generalizes to quaternions. If we identify the unit axis of rotation $\mathbf{a}$ with quaternion $\mathbf{a} = a_0 i + a_1 j + a_2 k$, we can express the rotation about axis $\mathbf{a}$ with angle α by quaternion

$$\exp\left(\mathbf{a}\frac{\alpha}{2}\right) = \cos\frac{\alpha}{2} + \mathbf{a}\sin\frac{\alpha}{2} \tag{2.2}$$

This can be proven by taking the Taylor series of $\exp(x)$, substituting $\mathbf{a}\frac{\alpha}{2}$ for x and considering that the quaternion product $\mathbf{a}\mathbf{a} = -1$ (because of the unit length of axis $\mathbf{a}$). Formula (2.2) defines the *exponential* mapping, which is a mapping from quaternions with zero scalar part (but not necessarily unit) to unit quaternions. The inverse mapping (*logarithmic*) is given as follows. If unit quaternion $\mathbf{q}$ is written as $\mathbf{q} = \exp\left(\mathbf{a}\frac{\alpha}{2}\right)$ (which is always possible), then $\log\mathbf{q} = \mathbf{a}\frac{\alpha}{2}$. Using these mappings, we can define the power of unit quaternion: $\mathbf{q}^t = \exp(t\log(\mathbf{q}))$. These concepts are essential for quaternion interpolation [29].

Conversions between 3×3 rotation matrix and unit quaternion can be derived from the quaternion product $\mathbf{q}\mathbf{v}\mathbf{q}^*$; see [32] for optimized conversion routines. Even though both quaternions $\mathbf{q}$ and $-\mathbf{q}$ represent the same rotation, their powers $\mathbf{q}_0^t$ and $(-\mathbf{q}_0)^t$ are different for $0 < t < 1$: one corresponds to clockwise and second to counterclockwise rotation, see Figure 2.2. During matrix to quaternion conversion, we can choose between $\mathbf{q}_0$ and $-\mathbf{q}_0$. This choice depends on each particular application. For example, in the case of blending, we typically want to select such signs so that the geodesic distances (i.e., the shortest paths on S_3) among the resulting quaternions are as small as possible. This can be done by an algorithm described independently in [105] and [66].

2.1.2 Dual Quaternions

Regular quaternions, even though very important in computer animation, have their limitations. In many algorithms, we need to work with both rotation and translation, whose
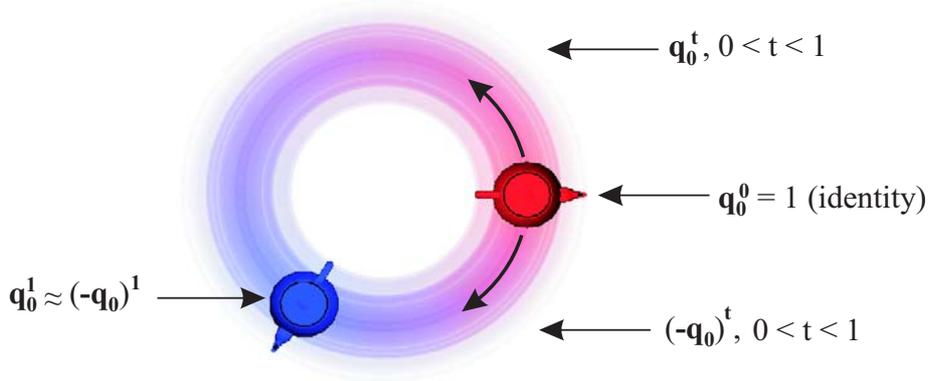
Figure 2.2: Quaternion antipodality. In this example, transformation of the teapot by $\mathbf{q}_0^t$ for $t \in [0, 1]$ produces a counterclockwise rotation (longer trajectory), while transformation by $(-\mathbf{q}_0)^t$ leads to a clockwise one (shorter trajectory).

composition is known as *rigid transformation*. The concept of homogeneous matrices is very useful, because it allows us to represent composition of rigid transformations by one 4×4 matrix multiplication. Unfortunately, this is not possible with regular quaternions. Dual quaternions can be understood as an extension of quaternion algebra that overcomes this limitation: unit dual quaternions represent 3D rigid transformations in the same way as regular quaternions represent 3D rotations.

Dual quaternions were developed by Clifford in the nineteenth century [26], along with the more general concept of geometric algebras. These algebras naturally contain not only vectors and quaternions, but also k-dimensional subspaces [133]. This leads to very elegant and dimension independent expressions, but unfortunately also to an increase in time and memory complexity [35]. Dual quaternions, in turn, are not so general, but are more compact and faster to manipulate. Some results in this thesis are based on the dual quaternion algebra, and therefore we summarize the most important facts in this section. Further information can be found in the books [94], [16] and [68].

Dual quaternions can be considered as quaternions whose elements are dual numbers. In the text, we distinguish dual quantities (i.e., scalars, vectors and quaternions) from non-dual ones by a caret. The algebra of dual numbers is similar to complex numbers: any dual number $\hat{a}$ can be written as $\hat{a} = a_0 + \epsilon a_\epsilon$, where a_0 is the non-dual part, a_ϵ the dual part and ϵ is a *dual unit* satisfying $\epsilon^2 = 0$. The dual conjugate is analogous to the complex conjugate: $\overline{\hat{a}} = a_0 - \epsilon a_\epsilon$. Multiplication of two dual numbers is given as $(a_0 + \epsilon a_\epsilon)(b_0 + \epsilon b_\epsilon) = a_0 b_0 + \epsilon(a_0 b_\epsilon + a_\epsilon b_0)$. The inverse of a dual number $\hat{a}^{-1}$ is given by

$$\frac{1}{a_0 + \epsilon a_\epsilon} = \frac{1}{a_0} - \epsilon \frac{a_\epsilon}{a_0^2}$$

as can be immediately verified. The previous expression is defined only when $a_0 \neq 0$. Purely dual numbers, that is dual numbers with $a_0 = 0$, do not have an inverse. This is

a fundamental difference from complex numbers, because every non-zero complex number has an inverse. The square root is defined only for dual numbers with a positive non-dual part, and it is computed as

$$\sqrt{a_0 + \epsilon a_\epsilon} = \sqrt{a_0} + \epsilon \frac{a_\epsilon}{2\sqrt{a_0}}$$

A dual quaternion $\hat{\mathbf{q}}$ can be written as $\hat{\mathbf{q}} = \hat{w} + i\hat{x} + j\hat{y} + k\hat{z}$, where $\hat{w}$ is the scalar part (dual number), $(\hat{x}, \hat{y}, \hat{z})$ is the vector part (dual vector), and i, j, k are the usual quaternion units. The dual unit ϵ commutes with quaternion units, for example $i\epsilon = \epsilon i$. A dual quaternion can be also considered as an 8-tuple of real numbers, or as the sum of two ordinary quaternions, $\hat{\mathbf{q}} = \mathbf{q}_0 + \epsilon \mathbf{q}_\epsilon$. Conjugation of a dual quaternion is defined using classical quaternion conjugation: $\hat{\mathbf{q}}^* = \mathbf{q}_0^* + \epsilon \mathbf{q}_\epsilon^*$

The norm of a dual quaternion can be written as $\|\hat{\mathbf{q}}\| = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \sqrt{\hat{\mathbf{q}}\hat{\mathbf{q}}^*}$, which expands to

$$\|\hat{\mathbf{q}}\| = \sqrt{\hat{\mathbf{q}}^*\hat{\mathbf{q}}} = \|\mathbf{q}_0\| + \epsilon \frac{\langle \mathbf{q}_0, \mathbf{q}_\epsilon \rangle}{\|\mathbf{q}_0\|}$$

The norm satisfies the multiplicative property $\|\hat{\mathbf{p}}\hat{\mathbf{q}}\| = \|\hat{\mathbf{p}}\|\|\hat{\mathbf{q}}\|$ (see [68]). The inverse of a dual quaternion is defined only when $\mathbf{q}_0 \neq \mathbf{0}$. In this case, we have $\hat{\mathbf{q}}^{-1} = \frac{\hat{\mathbf{q}}^*}{\|\hat{\mathbf{q}}\|^2}$. Unit dual quaternions are those satisfying $\|\hat{\mathbf{q}}\| = 1$. According to the previous formula, a dual quaternion $\hat{\mathbf{q}}$ is unit if and only if $\|\mathbf{q}_0\| = 1$ and $\langle \mathbf{q}_0, \mathbf{q}_\epsilon \rangle = 0$. Note that unit dual quaternions are always invertible (their inverse is just conjugation). We denote the set of unit dual quaternions as $\hat{Q}_1$. Geometrically, $\hat{Q}_1$ is a 6-dimensional manifold (called an image-space of dual quaternions [94]). Just like ordinary quaternions, dual quaternions are also associative, distributive, but not commutative.

As expected, unit dual quaternions naturally represent 3D rotation, when the dual part $\mathbf{q}_\epsilon = \mathbf{0}$. If we have a 3D vector $\mathbf{v} = (v_0, v_1, v_2)^T$, we define the associated unit dual quaternion as $\hat{\mathbf{v}} = 1 + \epsilon(v_0 i + v_1 j + v_2 k)$. The rotation of vector $\mathbf{v}$ by a dual quaternion $\hat{\mathbf{q}}$ then can be written as $\hat{\mathbf{q}}\hat{\mathbf{v}}\overline{\hat{\mathbf{q}}^*}$ (where $\overline{\hat{\mathbf{q}}^*}$ denotes both quaternion and dual conjugation). This is obvious: if $\mathbf{q}_\epsilon = \mathbf{0}$ then $\hat{\mathbf{q}} = \mathbf{q}_0$ and $\hat{\mathbf{q}}\hat{\mathbf{v}}\overline{\hat{\mathbf{q}}^*}$ simplifies to

$$\mathbf{q}_0(1 + \epsilon(v_0 i + v_1 j + v_2 k))\mathbf{q}_0^* = 1 + \epsilon \mathbf{q}_0(v_0 i + v_1 j + v_2 k)\mathbf{q}_0^*$$

where $\mathbf{q}_0(v_0 i + v_1 j + v_2 k)\mathbf{q}_0^*$ is the formula for rotation by an ordinary quaternion from Section 2.1.1.

An interesting fact is that dual quaternions can also represent 3D translations. A unit dual quaternion $\hat{\mathbf{t}}$, defined as $\hat{\mathbf{t}} = 1 + \frac{\epsilon}{2}(t_0 i + t_1 j + t_2 k)$ corresponds to translation by vector $(t_0, t_1, t_2)^T$ (note that dual quaternions work with *half* of the translation vector, in analogy to classical quaternions, which work with half of the angle of rotation). If we simplify $\hat{\mathbf{t}}\hat{\mathbf{v}}\overline{\hat{\mathbf{t}}^*}$, we obtain $1 + \epsilon((v_0 + t_0)i + (v_1 + t_1)j + (v_2 + t_2)k)$, which shows that the unit dual quaternion $\hat{\mathbf{t}}$ really performs translation by $(t_0, t_1, t_2)^T$. Rigid transformation is a composition of rotation and translation, and composition of transformations corresponds

to multiplication of dual quaternions. If the rotation is described by unit quaternion $\mathbf{q}_0$ and the translation by unit dual quaternion $1 + \frac{\epsilon}{2}(t_0 i + t_1 j + t_2 k)$ as before, then their composition is

$$(1 + \frac{\epsilon}{2}(t_0 i + t_1 j + t_2 k))\mathbf{q}_0 = \mathbf{q}_0 + \frac{\epsilon}{2}(t_0 i + t_1 j + t_2 k)\mathbf{q}_0 \tag{2.3}$$

We can verify by direct computation that the result is always a unit dual quaternion. Formula (2.3) tells us how to convert a (*quaternion*, *translation*) pair to a unit dual quaternion. Note that this conversion involves just computation of one regular quaternion product. The opposite conversion, from a unit dual quaternion $\mathbf{q}_0 + \epsilon\mathbf{q}_\epsilon$ to a (*quaternion*, *translation*) pair is equally easy. The rotation is just $\mathbf{q}_0$ and the translation is given as $2\mathbf{q}_\epsilon\mathbf{q}_0^*$.

Every unit dual quaternion $\hat{\mathbf{q}}$ can be written as

$$\hat{\mathbf{q}} = \cos\frac{\hat{\theta}}{2} + \hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2} \tag{2.4}$$

where $\hat{\mathbf{s}}$ is a unit dual vector with zero scalar part, see [94] or [30]. Note that this looks like the formula for ordinary quaternions, just employing the dual angle $\hat{\theta} = \theta_0 + \epsilon\theta_\epsilon$ and unit dual vector $\hat{\mathbf{s}} = \mathbf{s}_0 + \epsilon\mathbf{s}_\epsilon$. The geometric interpretation of those quantities is related to *screw motion*, that is a rotation and translation about the same axis. Chasle's theorem [30] states that any rigid transformation can be described by a screw motion, see Figure 2.3. Angle $\theta_0/2$ is the angle of rotation, and unit vector $\mathbf{s}_0$ represents the direction of the axis of rotation. $\theta_\epsilon/2$ is the amount of translation along vector $\mathbf{s}_0$, and $\mathbf{s}_\epsilon$ is the *moment* of the axis. Moment is an unambiguous description of the position of an axis in space. It is given by equation $\mathbf{s}_\epsilon = \mathbf{p} \times \mathbf{s}_0$, where $\mathbf{p}$ is a vector pointing from the origin to an arbitrary point on the axis. Which point we choose is not important, because for any other point of the axis, say $\mathbf{p} + c\mathbf{s}_0$ (where c is an arbitrary scalar), we obtain the same moment: $(\mathbf{p} + c\mathbf{s}_0) \times \mathbf{s}_0 = \mathbf{p} \times \mathbf{s}_0$. Computation of the moment and other screw parameters is hidden in the quaternion multiplication during conversion to a dual quaternion, Formula (2.3). We can also infer another insight: whereas classical quaternions can represent only rotations whose axes pass through the origin, dual quaternions can represent rotations with arbitrary axes.

Similarly as in Section 2.1.1, it is possible to derive the dual quaternion version of exponential mapping:

$$\hat{\mathbf{q}} = \exp\left(\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\right) = \cos\frac{\hat{\theta}}{2} + \hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2} \tag{2.5}$$

The logarithm of $\hat{\mathbf{q}}$ is thus $log(\hat{\mathbf{q}}) = \hat{\mathbf{s}}\frac{\hat{\theta}}{2}$. A power of a dual quaternion $\hat{\mathbf{q}}$ is then defined naturally: $\hat{\mathbf{q}}^{\hat{u}} = \exp(\hat{u}\log\hat{\mathbf{q}}) = \cos(\hat{u}\frac{\hat{\theta}}{2}) + \hat{\mathbf{s}}\sin(\hat{u}\frac{\hat{\theta}}{2})$. We see that the dual quaternion formulas are very similar to corresponding formulas for ordinary quaternions. However, it is necessary not to forget that in the former case, the numbers $\hat{u}$ and $\hat{\theta}$ are dual.
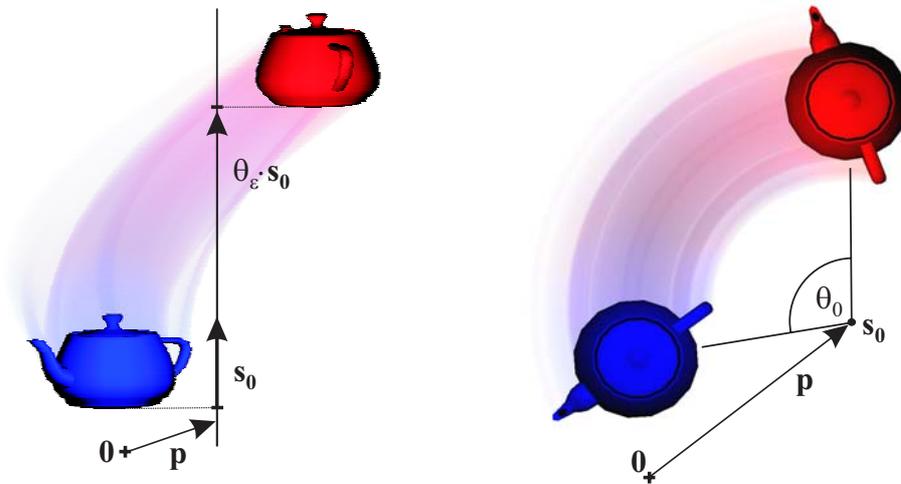
Figure 2.3: Example of a screw motion. Each rigid transformation can be described as a screw: rotation about an axis by angle θ_0 and translation with magnitude θ_ϵ along the same axis. The axis, with direction determined by unit vector $\mathbf{s}_0$, needs not pass through the origin $\mathbf{0}$. Its position in space is given by vector $\mathbf{p}$ pointing from the origin to some point on the axis. The choice of vector $\mathbf{p}$ is not unique, thus dual quaternions work with the unique moment: $\mathbf{p} \times \mathbf{s}_0$.

Dual quaternions inherit the antipodality of regular quaternions, i.e., both $\hat{\mathbf{q}}$ and $-\hat{\mathbf{q}}$ represent the same rigid transformation. This has the same consequences as in the case of classical quaternions, see Section 2.1.1. Namely, mapping from rigid transformations to unit dual quaternions is also one to two and the power $\hat{\mathbf{q}}^{\hat{u}}$ differs from $(-\hat{\mathbf{q}})^{\hat{u}}$ by the clockwiseness of the associated screw. The translation component of the motion is not affected.

2.2 Skin Deformation

In this thesis, we use the term *skinning* (equivalently *smooth skinning*, *matrix palette skinning*, *skeletal deformation* or *skeletal animation*) to refer to a skin deformation algorithm based on 1) a list L of rigid transformations, 2) reference mesh and 3) vertex weights, describing the relationship between L and mesh vertices. Note that this understanding of skinning has not been introduced until recently [63]. In previous literature, skinning is considered to be linked with a hierarchical structure of transformations (usually called *skeleton*). Even though this is intuitive (reminiscent to skeletons of real beings) it is also restrictive, as no skeleton is actually required in skinning, as shown in [63]. In the following, we will be therefore emphasizing this more general perspective.

Theoretically, any kind of skin deformation caused by an underlying skeleton can be called skeletal animation as well. For example, skeleton motion can drive a physically based

simulation of an elastic body, whose surface corresponds to the skin [23]. Although we survey such approaches in this section, they are not the topic of our further investigation.

2.2.1   Matrix Palette Skinning

Matrix palette skinning is the main objective of this thesis and therefore this section presents its basics in more details. An input model consists of the following data structures:

- **triangular mesh** M – any 3D object representing the rest-pose (undeformed) model. We assume there are m vertices in the mesh and we denote them as $\mathbf{v}_1, \ldots, \mathbf{v}_m \in R^3$.

- **list** L **of rigid transformations** – representing the current deformation. In certain special situations it will be useful to consider a more general class of transformations, e.g., affine ones. However, unless stated otherwise, only rigid transformations are assumed. We denote the length of list L as p and the individual transformations as $C_1, \ldots, C_p$ (often represented by 4×4 homogeneous matrices). Each transformation $C_1, \ldots, C_p$ influences part of the mesh M.

- **vertex binding** – describing which transformations from L influence which part of the mesh M. For each vertex $\mathbf{v}_k$ we have two lists of length $n_k \in Z$. The first list, $j_{k,1}, \ldots, j_{k,n_k} \in Z$, is a sequence of indices of transformations from L that influence vertex $\mathbf{v}_k$. Second list, $w_{k,1}, \ldots, w_{k,n_k} \in [0,1]$, is a sequence of vertex weights. Weight $w_{k,i}$ describes the amount of influence of transformation $C_{j_{k,i}}$ on vertex $\mathbf{v}_k$. Weights $w_{k,1}, \ldots, w_{k,n_k}$ must be convex, i.e., besides the non-negativity are required to satisfy $\sum_{i=1}^{n_k} w_{k,i} = 1$.

When matrix palette skinning is used in conjunction with a hierarchy of transformations (e.g., a skeleton), then $C_1, \ldots, C_p$ are rigid transformations corresponding to individual joints (nodes of the tree). In this case, the rigid transformation C_j represents translation and rotation of joint j from the rest-pose to the current (animated) posture. Formulas for computing the transformations $C_1, \ldots, C_p$ in this case are presented in [A.10]. The indices $1, \ldots, p$ of transformations $C_1, \ldots, C_p$ are often called *joints* (or *proxy-joints*) even if no actual skeleton is present.

There exist various software tools to design skeletally deformable models [9, 10]. Nevertheless, creation of a these model is a complex task requiring many hours of the animator's labor. Good models are seldom available for free, but can be purchased on-line [126] or together with a book [122]. Some simpler models might be equipped only with *rigid* skinning, which means that vertex weights are either 0 or 1 (which typically results in poor skin deformations). An automatic way to construct skinned animations is discussed in Chapter 7.

The basic operation with a skeletal model is to deform its skin according to a given list L of transformations. Note that both rest-pose mesh and vertex binding are constant,

as well as the length of the list L – only the individual transformations $C_1, \ldots, C_p$ vary during an animation. The most popular matrix palette skinning algorithm is linear blend skinning (LBS) [87]. With LBS, we assume that $C_1, \ldots, C_p$ are represented by homogeneous matrices; the deformed vertex positions $\mathbf{v}'_1, \ldots \mathbf{v}'_m$ are then computed as as follows:

$$\mathbf{v}'_k = \left(\sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}} \right) \mathbf{v}_k, \quad k = 1, \ldots, m \tag{2.6}$$

In situations where the vertex index will not be important, we drop the subscript k and write Formula (2.6) more concisely: $\mathbf{v}' = \left(\sum_{i=1}^{n} w_i C_{j_i} \right) \mathbf{v}$. In practice, we can exploit the distributivity of matrix products and rewrite

$$\left(\sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}} \right) \mathbf{v}_k = \sum_{i=1}^{n_k} w_{k,i} \left(C_{j_{k,i}} \mathbf{v}_k \right)$$

which explains why LBS is sometimes called *vertex blending*: it can be interpreted as blending transformed positions of $\mathbf{v}_k$. More advanced matrix palette skinning algorithms replace the linear combination of matrices, i.e., the term $\left(\sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}} \right)$, by a more sophisticated transformation blending method.

See Figure 2.4 for an example illustrating these concepts. There are two transformations in the figure, C_1 and C_2, corresponding to the transformations of shoulder and elbow joints from the rest-pose to an animated posture (note that C_1 is pure translation, as the shoulder is not rotated, but C_2 is a general rigid transformation). Vertex $\mathbf{v}_1$ is influenced solely by C_1, but $\mathbf{v}_2$ is influenced by both C_1 and C_2. Therefore, we have $n_1 = 1, j_{1,1} = 1, w_{1,1} = 1$ and $n_2 = 2, j_{2,1} = 1, j_{2,2} = 2, w_{2,1} = 0.5, w_{2,2} = 0.5$, assuming that $\mathbf{v}_2$ is influenced by both transformations equally.
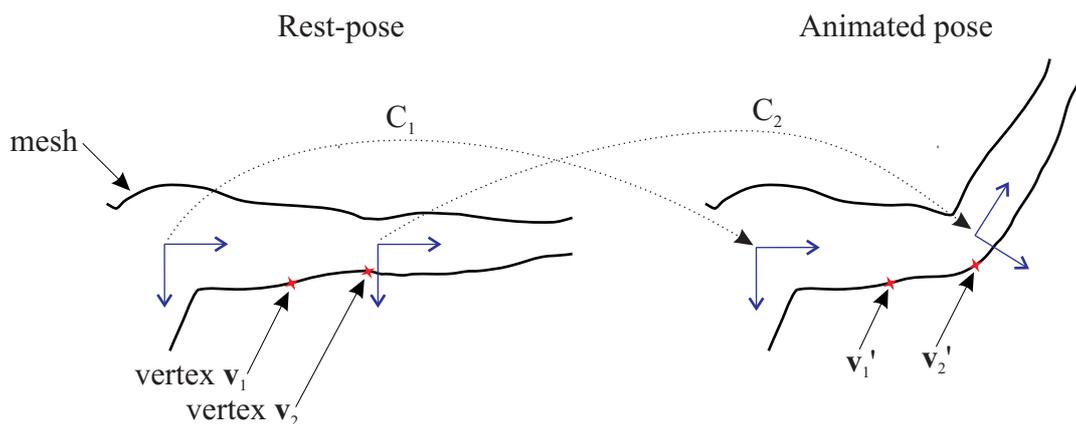
Figure 2.4: A silhouette of human arm animated by matrix palette skinning.

Although the LBS algorithm with properly weighted vertices produces smooth skin deformation, it does not look naturally for all joint transformations. It has been pointed out

already in [134] that LBS produces poor skin deformations when the joint rotations are large. A classic example of this behavior is known as a *candy wrapper* artifact. It is nicely obvious when the shoulder joint is twisted 180 degrees, see Figure 2.5, but it occurs also with not so extreme rotations [134].
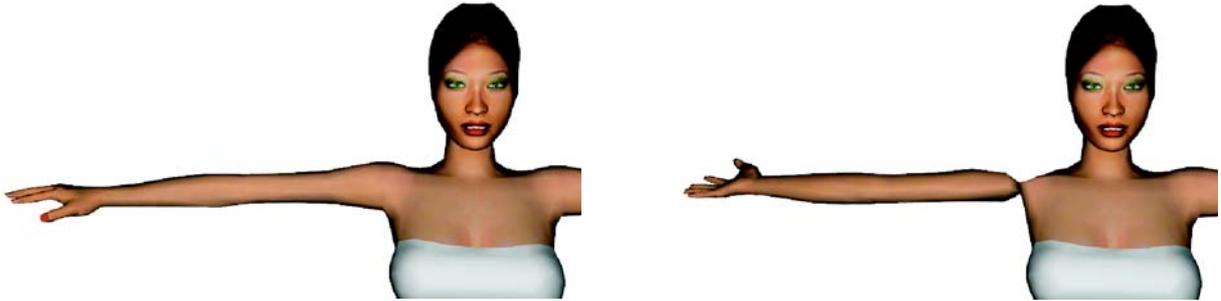
Figure 2.5: Left: rest-pose character model. Right: shoulder twisted by 180 degrees produces the candy wrapper artifact in LBS.

In spite of these problems, LBS algorithm is commonly used for real-time animation of virtual characters, especially in computer games. However, this might be only because there is no simple alternative.

2.2.2 Example Based Methods

An important class of skin deformation methods achieves better quality by using multiple input meshes. These meshes are often designed in extremal positions, where the artifacts of linear blending are most obvious. The problem of skin deformation can then be expressed as a problem of blending of the examples. The blending domain is the space of all transformations $C_1, \ldots, C_p$; some methods consider also additional parameters (e.g., gender, age, muscularity). Obviously, the blending space is a very high-dimensional one.

One of the first example based methods is pose space deformation [85]. It exploits the generality of the blending problem and provides a unified framework for both skeletal and facial animation. The basic idea is to use radial basis functions to blend example skins with different shapes. In [85], as well as in most other interpolation-based methods, only the *displacements* from LBS to the examples poses are blended, which considerably simplifies the problem. The blending function is a linear combination of radial basis functions, whose coefficients are derived using least-squares optimization. Unfortunately, no experimental results such as timings are reported in [85], but the authors claim that a real-time animation is possible (on the assumption of pre-computed coefficients).

A more advanced example-based method is presented in [120]. The most important difference to [85] is the introduction of *cardinal basis functions*, which play the role of a blending basis. Instead of computing a different function per each vertex as in [85], the functions are created as a linear combination of cardinal basis functions. According to [120], this results in orders of magnitude faster precomputation. The other improvements include a different form of radial basis function: sum of a linear component and a B-spline.

The usage of programmable graphics processing units (GPUs) for the example-based skin deformation is considered in [74]. The previous methods [85, 120] were not suitable for GPU execution, because they required a lot of memory to store the examples and the blending coefficients. This problem is tackled in [120] by applying singular value decomposition (SVD). The matrix of vertex displacements for all relevant poses is decomposed using SVD, which reveals the most important displacement components – called *eigen-displacements*. Only few most important eigen-displacements are necessary for a good approximation of the original shape. This results in considerable memory savings and enables to transfer the computations to the GPU.

Recently, a generalization of pose space deformation called weighted pose space deformation has been suggested [75]. This technique is useful also if only a limited number of examples is available. The main trick is that the distance function between two poses is no longer independent of the processed vertex, but depends on the vertex weights. This improves the interpolation of a sparse number of examples, but unfortunately also slows the process down. A real-time implementation on the current hardware is possible only by a smart utilization of the GPU, as suggested in [114]. This consists in executing the skin deformation algorithm in fragment shaders, in contrast to the straightforward solution utilizing vertex shaders [58]. This is done because fragment shaders offer higher parallelization and thus also faster run-time computations. However, the latest generation of graphics hardware (GeForce 8 Series) automatically assigns its stream processors to either vertex or fragment processing, so this trick should no longer be so important.

The advantage of the example-based methods is that they not only resolve the artifacts of LBS, but also provide higher control. For example, it is possible to achieve the muscle bulging effects easily, just by sculpting the contracted muscles. It is also possible to introduce additional parameters, such as gender, age, or muscularity, and blend them in the same framework as joint transformations. Thanks to this, it is possible to obtain interesting shape variations in run-time. The disadvantage is the necessity of acquiring the example skins, which can be costly. Such methods are therefore more popular in feature film animation (e.g., Shrek 2) than in real-time applications.

Other example based methods are not based on the concept of blending. This is the case of multi-weight enveloping [132], which is a direct generalization of LBS. It introduces more parameters and therefore greater flexibility to the deformation algorithm. Instead of one weight per transformation as in LBS, the multi-weight enveloping uses twelve – one per each entry of the homogeneous matrix. These numerous parameters are derived from examples using least squares optimization. This offers great versatility in run-time skin deformations,

comparable to example-based methods. However, this comes at a cost of complicated computation of the numerous weights. The user has to provide a sufficient number of examples, and the optimization process is reported to take several minutes. Another disadvantage is that while the LBS models can be weighted manually by animators [122], this is questionable with multi-weight enveloping. However, for high-fidelity characters, this is an appropriate method. The idea of multi-weight enveloping has recently been refined by Merry et al. [96], who proposed a linear framework supporting multiple weights per transformation matrix. Linearity has certain benefits: it is fast, it allows a simple least-squares optimization and it can be used to derive a measure of average distance across the space of poses. However, the example meshes are still necessary in order to obtain the weights – no direct way has been presented.

The example based algorithm discussed in [99] combines the advantages of [134] and [132]. The artifacts of LBS are resolved by adding extra joints, which are also exploited to simulate additional effects such as the muscle bulging. The vertex weights must be re-computed after the joint addition, which is done automatically using the provided examples. The vertex positions in the reference mesh are optimized together with vertex weights. An iterative optimization algorithm ensures that the deformed skin matches the provided examples as closely as possible (in the least squares sense).

2.2.3   Advanced Blending

As discussed in Section 2.2.1, the problems of LBS stem from the linear combination of transformation matrices. It is well known that this is a poor way of transformation blending. For example, if the input matrices are rigid transformations, the result of a direct linear combination can be an arbitrary matrix. This matrix can (and usually does) contain scale and shear components that are responsible for the LBS artifacts.

Therefore, a natural solution is to apply an advanced transformation blending method. One of the first algorithms from this category overcomes the artifacts by introducing auxiliary bones, in order to spread large rotations into smaller ones [134]. Even though this method reduces the artifacts, the vertex binding needs to be redesigned, which can be costly. Also, the accuracy of this method depends on the number of auxiliary bones, and finding an optimal trade-off between accuracy and efficiency requires some tweaking.

Another, more sophisticated method of transformation blending has been proposed by Alexa [3]. This method takes logarithms of the input matrices, computes their linear combination and exponentiates the result. The matrix exponential is defined as

$$e^A = \sum_{n=0}^{\infty} \frac{A^n}{n!}$$

and the logarithm of matrix B is matrix C such that $e^C = B$ (see [101]). Blending of matrix logarithms has better properties than direct linear combination. Specifically, it

preserves rigidity of input transformations, which prevents the candy wrapper artifacts. A drawback is the computational complexity of matrix logarithm and exponential.

Alexa's log-matrix blending has been applied to skinning by Magnenat-Thalmann et al. [89]. Due to efficiency constraints, they recommend to use the log-matrix blending only when the joint rotations are large and opt for linear blending otherwise. Unfortunately, it has been shown that log-matrix blending still has not optimal properties and in some cases may also produce unnatural skin deformations, see Chapter 4 and [13]. Solutions of these limitations have been discussed in [86] and [53], however, only for the case of transformation interpolation.

Another skinning method based on advanced blending has been proposed by Jim Hejl from Electronic Arts [56]. His idea is to convert the rotation parts of the skinning matrices to quaternions and blend them linearly. Because of the properties of quaternion blending, this produces nice skin deformations in many situations. Unfortunately, this approach ignores the translational part of the skinning transformations, i.e., the deformed vertex is only allowed to rotate around a given fixed point. This works perfectly in cases where the influencing bones share a common joint. However, for more complex joint influences, e.g., those usually occurring around the armpit or dorsum of character models, artifacts result from the fact that the center of rotation is fixed and does not adapt to the actual posture (as noted already in [58]).

Yang's skinning approach [135] also falls into the category of advanced blending methods. In this approach, a rotation between successive limbs is decomposed into twist and swing components, and both are blended independently. In addition to this, the joint collapsing artifact of linear blending is remedied by a stretching operator, which pushes the skin outwards the problematic point. However, this method probably requires some non-trivial tweaking of the stretching operator parameters. The article also does not discuss how to blend rotations among more than two joints.

Another geometric skin deformation algorithm is presented in [14]. It is based on a convolution with a complex auxiliary structure – a medial. This algorithm did not become quite popular, since the computation of medial is non-trivial and the method is patented. Another skin deformation algorithm, which introduces novel skeleton-skin binding method, is presented in [60]. Their algorithm is based on sweep surfaces of moving ellipses and produces plausible deformations of human characters. However, this comes at the cost of a more complex (and non-standard) rigging method.

2.2.4   Other Skin Deformation Methods

Numerous algorithms have been developed especially for the case of virtual humanoids. They use more or less advanced knowledge about human anatomy, and therefore are not applicable to general skeletally deformable models. The basic idea of these methods is to use a layered decomposition of the virtual human to skeleton, muscles, fat tissues and skin [67]. All these layers contribute to the final shape of the skin. The most important difference

to skeletal animation is the introduction of muscles, that can be modelled, e.g., with *metaballs* [67]. Metaballs smoothing can be used to mimic fat tissues. In order to achieve a real-time animation, the final model is converted into *cross-section* representation [125, 64]. The cross-sections (or contours) exploit the fact that the human body can be decomposed into a collection of cylindrical objects: torso, arms and legs (while the head, hands and feet are animated separately). The cross-section based animation has also been used in the context of H-ANIM compliant models [11].

When considering only virtual humans, it is appealing to simulate the human anatomy directly. The visually most important elements are the muscles, which can be modelled using control curves [7, 137]. A control curve is attached to bones (in analogy to real muscles) and animated either geometrically using B-splines [137], or physically by a mass-spring system [7]. The muscle is deformed by its control curve, which moves itself according to the skeleton posture. The muscle motion then interacts with the fat layer, which is simulated as linearly elastic material. This layering is able to create highly realistic effects. Such methods are however not much favoured by animators, because they usually prefer direct control of the animated model [106]. Moreover, muscle design is a tedious process, even though this can be ameliorated using more advanced tools [8].

Another physically based approach models the interior of an object as an elastic material (without assuming any anatomical structure). This can be solved using the finite element method [23, 48] even at interactive framerates. The advantage of this approach is a realistic deformation and an automatic inclusion of secondary (inertial) animation effects. The main drawbacks are time consuming computation and difficult direct control. Note that dynamic effects can be added also to matrix palette skinning, as discussed in [78].

Free-form deformation (FFD) is a general deformation technique originally proposed in [116]. The usage of FFD for character skinning has been discussed in [119], proposing so called *surface-oriented* FFD, which is driven by a coarse version of the skin. This coarse mesh can be deformed by any simpler skin deformation algorithm, typically LBS.

2.3   Collision Detection

While in the real world the objects usually can not penetrate one another, the contrary is true in virtual reality. In order to create an illusion of the real world, it is necessary to simulate the physics of colliding objects. The first, and usually the most time-consuming step in this task is collision detection (CD). There exist several classes of CD algorithms. The basic one is *static* CD, where the objects are assumed to be rigid, i.e., transformed only by rotation and translation. A more challenging problem arises when considering deformable objects, such as those animated using matrix palette skinning (Section 2.2). *Continuous* (or dynamic) CD algorithms present another important class of algorithms, which take into account the continuous motion of simulated objects. A different benefit

offer *time-critical* algorithms, which can be interrupted and asked for an approximate solution.

2.3.1   Static Collision Detection

The problem of static collision detection is fairly well studied, see [65] for a survey. In this short overview, we focus mainly on the static CD algorithms based on a bounding volume hierarchy (BVH). The BVH-based algorithms are quite efficient and do not pose any restrictions on the input data (such as convexity or zero genus). This makes them appealing for many applications.

The input of a basic CD algorithm are two triangular meshes M_0, M_1 (sometimes called triangular soups, to indicate that no topological structure is assumed). We need to check if there exists $u_0 \in M_0$ and $u_1 \in M_1$ such that u_0 intersects u_1. A naive static CD algorithm could test each pair from $M_0 \times M_1$. Such algorithm would have time complexity $O(|M_0||M_1|)$, which is unacceptable even for models with relatively small number of triangles (i.e., thousands), since CD usually has to be computed in real-time.

It is possible to speed up CD using BVHs. The *bounding volume* of a set of triangles T is denoted as $B(T)$ and it can be any convex subset of R^3 such that $\forall t \in T : t \subseteq B(T)$. Obviously, the smallest bounding volume is the convex hull. In practice, only approximations of the convex hull are used. The classic bounding volumes are spheres [108], Axis-Aligned Bounding Boxes (AABBs) [128], Oriented Bounding Boxes (OBBs) [39], and k-Discrete Oriented Polytopes (k-DOPs) [72]. More recent ones include QuOSPO-trees [54] (based on k-DOPs), Boxtrees [136] and Sphere Swept Bounding Volumes [79]. AABB is a box with edges parallel to the coordinate axes of some fixed coordinate system (usually the world coordinate system). OBB is an arbitrarily oriented box and k-DOP ($k \in Z$ even) is a convex polytope, whose face normals come from a fixed set of $k/2$ orientations. Note that if the first three orientations correspond to the coordinate axes, then 6-DOP is equivalent to AABB.

The pre-processing step of a BVH-based CD algorithm is to construct a tree of the chosen type of bounding volumes for a given triangular mesh M. Each node of this tree corresponds to some set of triangles, for example the root corresponds to the whole M. Extending our notation, $B(u)$ can be interpreted as a bounding volume of the set of triangles corresponding to node u. For conciseness, only binary trees are considered in the following; generalization to n-ary trees is straightforward. Let us denote the left child of node u as $l(u)$ and the right child as $r(u)$. The bounding volume tree can be constructed in a top-down manner, according to Algorithm 2.1. An example of a sphere tree constructed by this algorithm is in Figure 2.6.

The splitting rule mentioned in step (3) of Algorithm 2.1 is typically only a simple heuristics. An efficient one is to construct the smallest enclosing AABB and divide it along its longest side into two equal halves. The triangles are then assigned to either part. Other heuristics were studied in [128], but none of them offered better results. General collision
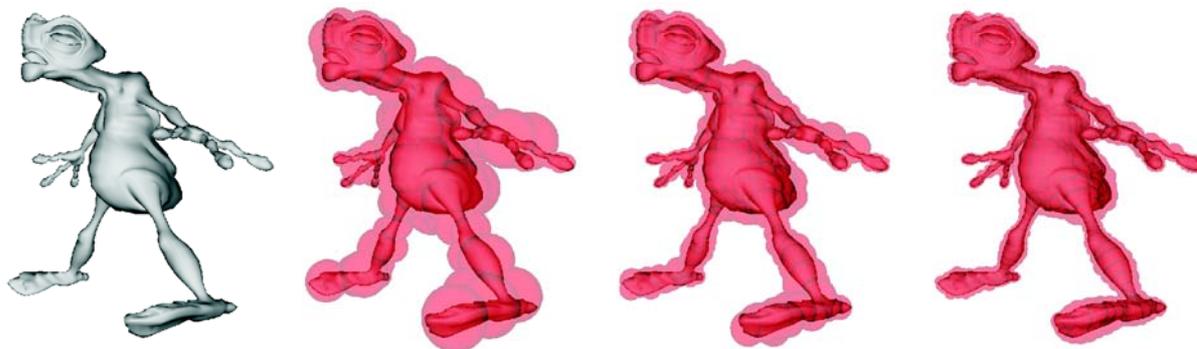
Figure 2.6: Three levels of the sphere tree constructed by Algorithm 2.1.

Algorithm 2.1: Construction of a bounding volume hierarchy
Input: Triangular mesh M
Output: Root of the created tree
BUILDTREE(M)
(1)   create node p
(2) $B(p) = B(M)$
(3)   split M according the chosen splitting rule into M_l and M_r
(4)   **if** $(M_l = 0)$ **or** $(M_r = 0)$
(5) $l(p) = r(p) = $ nil
(6)   **return** p
(7) $l(p) = $ BUILDTREE(M_l)
(8) $r(p) = $ BUILDTREE(M_r)
(9)   **return** p

detection algorithm based on a BVH is independent of the actual bounding volumes type: see Algorithm 2.2.

The function LEAF used by the algorithm just checks if its parameter is a leaf node. The function DISJOINT tests whether two bounding volumes are disjoint. However, the only condition we require from the function DISJOINT is that if it returns positive answer, then the bounding volumes must be disjoint. It is acceptable if the function returns negative answer even for disjoint bounding volumes. This is known as *conservative* testing [72] and can potentially produce redundant tests, but not an incorrect result. In spite of the redundant tests, the conservative approach can result in a faster CD algorithm than exact (and slower) intersection tests.

Algorithm 2.2: Collision detection using a bounding volume hierarchy
Input: Roots r_0, r_1 of BVHs
Output: Yes/no answer if the meshes represented by r_0, r_1 collide
COLTEST(r_0, r_1)
(1)   **if** DISJOINT$(B(r_0), B(r_1))$
(2)   **return** NO
(3)   **else**
(4)   **if** (LEAF(r_0) **and** LEAF(r_1))
(5)   test all triangles from r_0 against all triangles from r_1
(6)   **return** YES iff any pair intersected
(7)   **else if** (LEAF(r_0) **and not** LEAF(r_1))
(8)   **return** COLTEST$(r_0, l(r_1))$ **or** COLTEST$(r_0, r(r_1))$
(9)   **else if** (**not** LEAF(r_0) **and** LEAF(r_1))
(10)   **return** COLTEST$(l(r_0), r_1)$ **or** COLTEST$(r(r_0), r_1)$
(11)   **else**
(12) $r_i =$ the node with larger volume of $B(r_i)$, $i = 0, 1$
(13)   **return** COLTEST$(l(r_i), r_{1-i})$ **or** COLTEST$(r(r_i), r_{1-i})$

2.3.2 Deformable Collision Detection

In this thesis, we are especially interested in CD with skeletally deformable objects, i.e., objects whose shape changes during animation. The methods from Section 2.3.1 are not directly applicable to this situation, because they assume rigid bodies.

One of the first approaches to deformable CD is based on a complete BVH refitting, which adapts the bounding volumes to new vertex positions. The bottom-up refitting first recomputes the bounding volumes in the leaves. In the subsequent steps, bounding volumes in the parent nodes are refitted so that they enclose bounding volumes of their children. This reduces the time complexity from $O(n^2)$ required for the tree construction (Algorithm 2.1) to $O(n)$, where n is the number of triangles. However, the refitting can produce a sub-optimal BVH, because the topology of the hierarchy is unchanged. This is outweighted by the fact that refitting is about 10-times faster than rebuilding (as reported for the case of AABBs in [128]) so it is the algorithm of choice in practice.

The original bottom-up refitting algorithm is improved in [80] by combining bottom-up and top-down update and by using higher order trees instead of binary ones. Recently, this has been further optimized by exploiting temporal coherency [82, 83]. The refitting of a sphere tree for deformable object is studied in [20], using approximate enclosing spheres. Their algorithm optimizes the refitting procedure by considering only the vertices with non-zero displacement. Although this optimization helps in applications with localized deformations, it is of little use when all vertices are displaced (as is typically the case in skeletal deformation). Note also that bottom-up refitting of a sphere tree produces

conservative bounding spheres, whereas refitting of AABBs or k-DOPs gives an optimal one [128, 62].

The advantage of the complete BVH refitting is its generality – the method does not depend on the actual deformation model, because it works directly with the displaced vertices. The drawback is in the time-complexity: many bounding volumes may be refitted uselessly, because the subsequent CD query usually needs only some bounding volumes. In the case of the full BVH refitting, the time complexity is at least linear in the number of vertices. A more efficient strategy is to exploit the properties of a specific deformation model (if available). This makes possible to refit only those bounding volumes that are really necessary for the collision detection. This has been successfully implemented for models deformed by linear morphing [81]. This is a rather simple deformation model based on linear combinations of mesh vertices. In this deformation model, the bounding volumes can be simply deformed by the same algorithm as the object itself. However, the bounding volumes produced this way can be conservative. This is improved in [73], where a construction of tight bounding volumes for linear morphing is presented.

Another interesting deformation model are so called *reduced coordinates*, based on linear combinations of displacement fields [62]. If $\mathbf{p}$ is a vector of undeformed point locations, columns of matrix U describe the displacement fields and $\mathbf{q}$ are the reduced coordinates, then the deformed point locations $\mathbf{p}'$ are computed as

$$\mathbf{p}' = \mathbf{p} + U\mathbf{q}$$

An efficient on-demand refitting operation for this model is described in [62]. Although the displacement fields (columns of matrix U) can describe any kind of deformations, their combination is linear and therefore unsuitable for skeletally deformable models.

Other important approaches to deformable CD are not based on a BVH. For example, the bucket-tree algorithm [37] assumes a division of space instead of the object. The space is divided using an octree, whose final levels are equipped with buckets for individual primitives (typically triangles). When the primitives move or deform, they are shifted among individual buckets. The CD algorithm then considers only pairs of primitives sharing the same bucket, while the others can be culled. The space division paradigm is also applied in spatial hashing, presented in [123].

The constant advances in graphics hardware enable to exploit their overwhelming computational power also for deformable CD. This idea is pursued by image-space techniques [70, 55, 44, 42, 43]. The basic trick is to perform an occlusion query (an operation natively supported by the GPU). The occlusion query works by rasterizing the first object, thus filling the Z-buffer. Then the second object is rasterized: if all its pixels pass the Z-test, it means that the whole object is in front of the first one and thus there is no collision.

The advantage of these methods is that they do not need any pre-processing (and therefore are also more memory efficient). Their disadvantage is the necessity of either hashing or rasterizing all primitives, which disables sub-linear execution time (possible with BVH-based

methods). Moreover, the image-space methods have their accuracy limited by resolution of the image plane, although in [42], an exact image-space algorithm is presented. The exactness is achieved by inflating objects (formally: using the Minkowski sum), so that no collisions are missed due to the limited precision of the image plane.

2.3.3   Other Problems in Collision Detection

In spite of the importance of static and deformable collision detection, there exist other important problems in CD: continuous, time-critical and self-collision detection. This section overviews the methods of solving these more advanced collision detection problems.

The input of continuous CD are two objects represented by triangular meshes, description of their motion and a time interval. The task of the continuous CD is to find out whether the triangular meshes intersect during the given time interval and if yes, report the time of their first contact (in general, there can be more collisions during the time interval). A suitable model of in-between motion is essential for continuous CD. Typically, each object is assumed to undergo a screw motion (see Section 2.1.2). On the assumption of screw motions, an algebraic solution is possible [109]. This method reduces the problem to third degree polynomials which can be solved analytically. The final continuous CD system is presented in [110] and works by first performing continuous CD queries on bounding spheres, followed by continuous CD tests of individual primitives (i.e., vertex-edge and edge-edge tests).

A different approach to continuous CD is based on interval arithmetics [121], which is a general tool to construct bounds of (possibly multidimensional) functions. A continuous CD method based on interval arithmetics and a BVH of oriented-bounding boxes is presented in [111]. The bounds produced this way need not be tight in general, which leads to conservative tests. However, interval arithmetics can be used in more general situations than the algebraic solution, for example in the case of articulated kinematic chains [112, 113] (i.e., linkages of inter-connected rigid bodies).

A problem common to both static and continuous CD are the fluctuations of time complexity: in certain cases, the CD query needs a considerable amount of time (e.g., in close proximity situation), while in other cases it may terminate quickly (e.g., for objects far away). Clearly, this is very disadvantageous for real-time applications, which would ideally need algorithms with constant time complexity. *Time-critical* CD offers a solution to this problem, which is based on the fact that imperfect results are more acceptable than lagging (caused by waiting for an exact solution). The time-critical CD introduced in [59] is based on a hierarchy of bounding spheres. The object is approximated by a union of spheres, each level of the hierarchy standing for a higher level of detail. The CD Algorithm 2.2 is slightly altered, so that before entering the next level of the hierarchy, the collisions on the previous level must already be fully resolved (i.e., breadth-first search is used instead of depth-first). This makes possible to interrupt the algorithm and ask for an approximate solution (which is actually an exact solution for a union-of-spheres approximation of the

object). The main problem is to construct a good approximation with a small number of spheres. Modern algorithms exploit dynamic and adaptive medial-axis approximation in order to create efficient sphere trees [17, 18]. Recently, a time-critical algorithm supporting deformable objects has been discussed [95], but this area is still open to further investigation.

Self-collision detection, i.e., detection of collisions of an object with itself, is especially important in cloth simulation. Even though it is correct to simply execute Algorithm 2.2 on two identical objects, it is not very efficient. An object is inevitably in close-proximity with itself and thus culling of non-colliding parts becomes more difficult. In order to achieve an efficient self-CD algorithms, special techniques must be applied. For example, the method presented in [88] optimizes CD by bounding normals of each sub-piece of cloth. If there exists a direction $\mathbf{v}$ such that all normals within the sub-piece have a positive dot product with $\mathbf{v}$ and the projection to the plane orthogonal to $\mathbf{v}$ is collision free, than the sub-piece of cannot contain any self-collisions. This rule enables an efficient culling even in the case of self-collision detection. Another optimization for the cloth CD is presented in [97]. Their algorithm is based on an oriented inflation of k-DOPs which helps to bound the motion of the cloth. A more general family of methods is based on mesh decomposition [41, 40]. This reduces the problem to the standard n-body CD and thus enables to use efficient GPU-based algorithms. Another GPU-oriented approach to self-CD is presented in [25]. However, this method performs no culling and is thus useful only for relatively small models (about 1000 triangles).

2.4   Skinning Mesh Animations

The idea of using skinning as a method of data reduction is relatively new (being introduced in 2005) [27, 63, 91]. However, already before, a similar problem has been tackled in the context of clothed virtual humans: Oh et al. [102] present a system for the automatic dressing of a virtual human that is animated by matrix palette skinning. Alternatively, Cordier and Magnenat-Thalmann [28] propose to simulate cloth far from the body using a simplified physical model. Even though real-time, the physical simulation is reported to consume a lot of CPU time, which is a disadvantage over the computationally cheap skeletal animation.

Automatic construction of skinned approximations can be considered as an animation compression method. Various animation compression algorithms have been described since the pioneering work of Lengyel [84]. Alexa proposes applying Principal Components Analysis (PCA) [4]. PCA can also be advantageously augmented by another common compression technique, Linear Prediction Coding, as shown in [69]. Decorrelation based on wavelets also has been successfully applied to animation data [49]. Sattler et al. [115] show the advantages of clustered PCA and present fast GPU-based decompression. Another interesting idea is to encode the mesh as *geometry images* [47] and apply established 2D animation compression methods [19]. Alternatively, deformable geometry can be stored

using progressive multiresolution meshes [71], which represent the animation at multiple levels of detail. This allows the correct level of detail to be selected at runtime, either in a static or view-dependent way.

While data reduction is important, it is not the only issue in interactive 3D applications. Another tasks are equally important, such as efficient hardware accelerated rendering and collision detection. For example, skinned approximations can be exploited to speed up collision detection and facilitate rest-pose editing [63]. Also, it should be emphasized that skinning is not a linear representation. In fact, skinning can easily outperform even the best linear representation (given by Principal Component Analysis). This is because skinning allows objects to rotate and/or bend. For example, Principal Component Analysis does not work very well on a rotating rigid object (unless the rotation is small and the trajectories can be approximated well by straight lines). However, rotating a rigid object presents no problem for skinning, which needs just one proxy-joint to represent such an animation exactly.

Besides Skinned Mesh Animations (SMAs) [63], other techniques for approximating animations by skinning have been described. Collins and Hilton [27] propose a method based on a rigid transformation basis. Their algorithm delivers efficient data reduction, but the reconstructed animation suffers from discontinuities between individual clusters. The authors suggest correcting this by adding another post-processing step to smooth out the geometry. Mamou et al. [91] present a variation of SMAs with emphasis on animation compression. However, all of these algorithms focus only on quasi-articulated animations and are not suitable for highly deformable ones, such as those of cloth or elastic materials.

3 Spherical Blend Skinning

Spherical blend skinning (SBS) is our first attempt to resolve the artifacts of linear blend skinning (LBS) (see Figure 3.1). Even though in Chapter 4 we show that spherical blending is not as efficient as our dual quaternion based approach, we believe that SBS still deserves to be mentioned due to the following reasons. First, the ideas behind SBS reveal certain insights into the geometrical background of the problem, which help to understand the subsequent dual quaternion method (in our opinion, it is not a coincidence that SBS has been discovered before the dual quaternion skinning). Second, for SBS exists an efficient collision detection algorithm (Chapter 6), while this is an open problem for dual quaternion skinning.
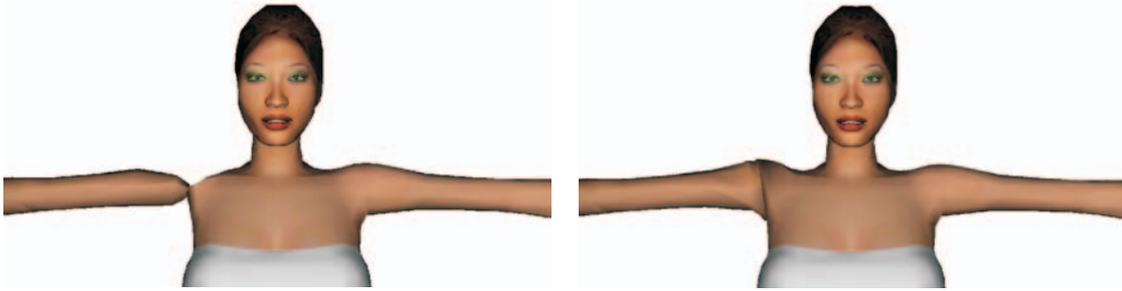
Figure 3.1: Left: the candy wrapper artifact of linear blend skinning, right: the same posture deformed by spherical blend skinning.

The problems of LBS arise from the fact that Formula (2.6) performs linear blending of matrices, which is known to be a poor way of transformation blending [3]. Specifically, even though the transformations C_{j_i} are rigid, their linear blend ($\sum_{i=1}^{n} w_i C_{j_i}$) needs not be rigid. The basic idea of SBS is to fix the skinning by employing a more advanced blending method. We hypothesize that if the applied blending algorithm preserves rigidity of input transformations, artifacts such as the candy wrapper will disappear. This hypothesis has proven to be true, see Figure 3.1. Intuitively, the blending of vertices $C_{j_i} \mathbf{v}$ does not operate on a line segment as LBS, but rather on a spherical arc, see Figure 3.2.

3.1 Beyond Linear Blending

Obviously, the problem of blending matrices C_{j_i} is caused by their rotational parts, because the group of rotations is not a linear space. If we want to avoid artifacts, we must ensure that the blending method always returns a correct rotation. This is possible even when working with rotation matrices, by finding the closest orthonormal matrix [12]. Unfortunately, this is a rather time-consuming operation, because as much as 6 orthonormality constraints must be satisfied (a rotation matrix has 9 elements, but 3D rotation has only 3
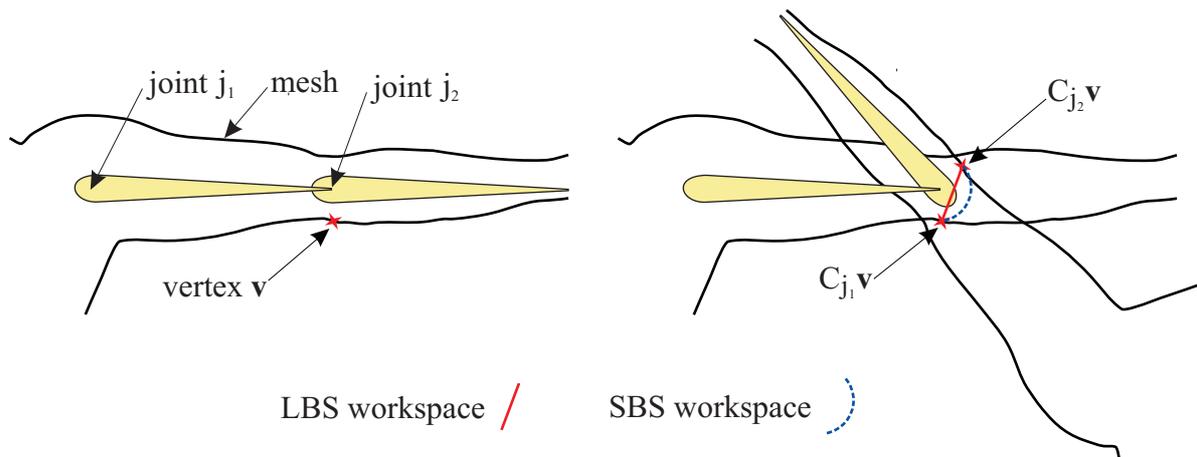
Figure 3.2: The set of possible results of LBS is a line segment, while SBS gives a circular arc.

degrees of freedom). The quaternion algebra (Section 2.1.1) is more advantageous for this purpose, having only one constraint and thus allowing a very efficient normalization.

However, there is one more (and perhaps even more serious) problem with this approach: it is necessary to select a suitable center of rotation. We demonstrate this problem on an example of human arm (see Section 4.1 for a more formal treatment). Consider that the arm geometry is influenced by two joints j_1 and j_2, such as in Figure 3.2. The transformation of the whole mesh by C_{j_1} is illustrated in the leftmost column of Figure 3.3 and the transformation of the same geometry by C_{j_2} in the rightmost column (note that the results are identical in both rows of these columns). The columns in the middle show the progress of interpolation between C_{j_1} to C_{j_2}. The only difference between the two rows in Figure 3.3 is in the choice of the center of rotation. In the top row, the rotation center $\mathbf{r}_c$ is set to the position of joint j_2 (elbow) in the reference posture. Note that $C_{j_1}\mathbf{r}_c = C_{j_2}\mathbf{r}_c$, therefore also the transformed rotation center is constant during the interpolation. In the bottom row of the figure, the rotation center $\bar{\mathbf{r}}_c$ is set to the position of joint j_1 (shoulder) in the reference posture. Since $C_{j_1}\bar{\mathbf{r}}_c \neq C_{j_2}\bar{\mathbf{r}}_c$, the transformed rotation center is linearly interpolated from $C_{j_1}\bar{\mathbf{r}}_c$ to $C_{j_2}\bar{\mathbf{r}}_c$. By comparison with the starting mesh (drawn gray in each frame), it is obvious that the center of rotation choice in the top row is much more advantageous. In this case, the interpolation of every single point is a circular arc (as in Figure 3.2), whereas a disturbing drift is inherent to any other choice of rotation center (such as $\bar{\mathbf{r}}_c$).

Unfortunately, the condition of zero translation cannot be always satisfied, typically for more than two influencing joints. However, it is possible to *define* the rotation center as the point whose transformations by associated matrices are as close as possible. This minimizes the drift and thus mimics the natural skin behavior, i.e., the avoidance of an excessive stretching. We find the center of rotation $\mathbf{r}_c$ as the least-squares solution of the
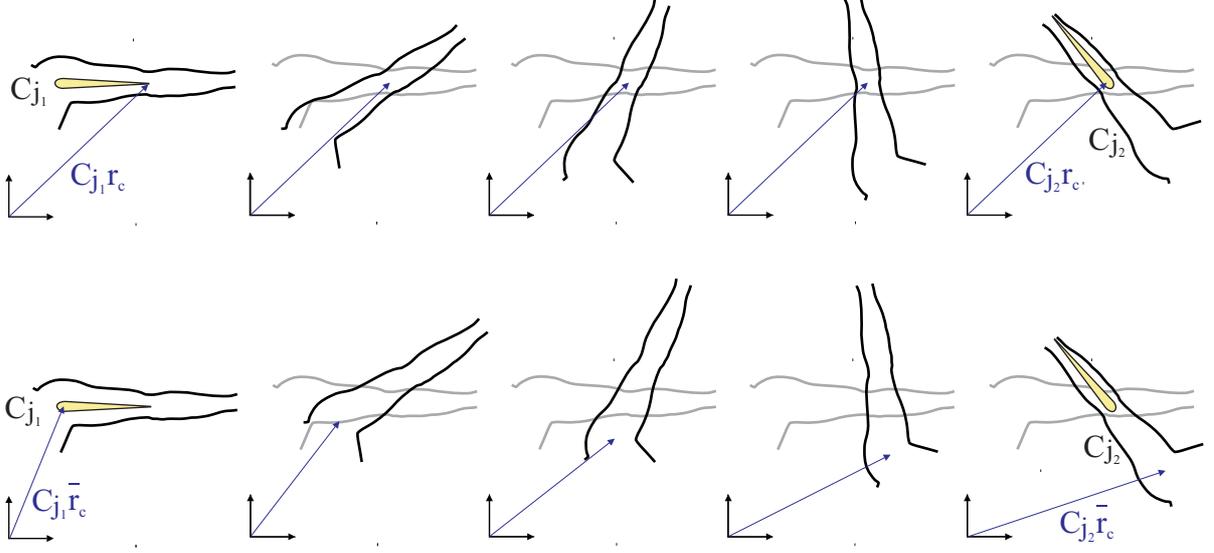
Figure 3.3: The correct center of rotation is chosen in the top row, while a sub-optimal in the bottom one. In the middle columns, notice the difference of the elbow position with respect to the original skin.

system of $\binom{n}{2}$ linear vector equations

$$C_a\mathbf{r}_c = C_b\mathbf{r}_c, \ \ a < b, \ \ a, b \in \{j_1, \ldots, j_n\} \tag{3.1}$$

Each homogeneous matrix C_i has structure

$$C_i = \left(\begin{array}{cc} C_i^{rot} & \mathbf{C}_i^{tr} \\ \mathbf{0}^T & 1 \end{array} \right)$$

where C_i^{rot} is a 3×3 orthogonal matrix and $\mathbf{C}_i^{tr}$ is a translation vector. This enables us to re-write the linear system to

$$
\begin{aligned}
C_a^{rot}\mathbf{r}_c + \mathbf{C}_a^{tr} &= C_b^{rot}\mathbf{r}_c + \mathbf{C}_b^{tr} \\
(C_a^{rot} - C_b^{rot})\mathbf{r}_c &= \mathbf{C}_b^{tr} - \mathbf{C}_a^{tr}
\end{aligned}
$$

If we stack all these equations to one matrix D and the right-hand sides to vector $\mathbf{e}$, we can write the whole system as

$$D\mathbf{r}_c = \mathbf{e}$$

where D is a $3\binom{n}{2} \times 3$ matrix, $\mathbf{r}_c$ is a 3-dimensional unknown vector and $\mathbf{e}$ is $3\binom{n}{2}$-dimensional vector. In general, we cannot make any assumptions about the rank of matrix D, which can vary from 0 to 3 (consider for example $n = 2$ and $C_{j_1} = C_{j_2}$). Therefore, we search the optimal solution $\mathbf{r}_c$ in least-squares sense. If there are multiple solutions giving the minimal $\|D\mathbf{r}_c - \mathbf{e}\|$, the $\mathbf{r}_c$ with the minimal norm is chosen. This can be done in a robust way using the singular value decomposition (SVD), followed by computation of pseudo-inverse matrix. To perform these computations, we use the LAPACK software [5].

Even though LAPACK routines are efficient, computation of the center of rotation per each vertex would not result in a real-time algorithm. Fortunately, the center of rotation depends only on the transformations of joints $j_1, \ldots, j_n$ and not the vertex itself. Therefore, if we encounter another vertex assigned to the same set of joints $j_1, \ldots, j_n$, we can re-use the center of rotation computed formerly (cached). Moreover, if there is only one, or two neighboring joints that influence the vertex, we can determine the center of rotation explicitly and omit the SVD computation completely. It turns out that the number of different non-trivial joint sets, and therefore the number of running the SVD, is surprisingly small for common models – about several tens (see Section 3.4). This enables the real-time performance.

However, as we discuss in Section 4.1, the caching of rotation centers introduces some limitations. This is because vertex weights are not considered in Formula (3.1). Theoretically, it would be much more natural, if, for example, a transformation with weight 0.1 did not influence the center of rotation by the same amount as a transformation with weight 0.9. Unfortunately, it is not affordable to execute SVD per each vertex, so it is necessary to settle for the uniform centers of rotations.

3.2 Rotation Blending

The problem of rotation blending has already received some attention [22, 105], as well as the problem of blending multiple general transformations [3]. Unfortunately, it has been shown later that Alexa's method [3] does not work very well for rotations [13]. More accurate (rotation specific) methods [22, 105] do not have such problems, but are substantially slower then the simple linear interpolation used in LBS. Since our goal is an algorithm with comparable time complexity as LBS, we propose an approximate but fast quaternion linear blending (QLB). For the case of two rotations, we compare this method with the established SLERP.

Recall that a rotation around axis $\mathbf{a}$ (unit length vector) with angle 2α corresponds to quaternion $\mathbf{q}' = \cos\alpha + \mathbf{a}\sin\alpha$. However, this correspondence is not unique, because both quaternions $\mathbf{q}'$ and $-\mathbf{q}'$ represent the same rotation (see Section 2.1.1). The SLERP of two unit quaternions $\mathbf{p}, \mathbf{q}$ assumes that their dot product $\langle\mathbf{p}, \mathbf{q}\rangle \geq 0$. If the dot product $\langle\mathbf{p}, \mathbf{q}\rangle < 0$, we use $-\mathbf{p}$ instead of $\mathbf{p}$, which is possible because both $\mathbf{p}$ and $-\mathbf{p}$ represent the same rotation. The SLERP of $\mathbf{p}, \mathbf{q}$ with interpolation parameter $t \in [0, 1]$ is given by the following formula (see [32]).

$$SLERP(t; \mathbf{p}, \mathbf{q}) = \frac{\sin((1-t)\theta)\mathbf{p} + \sin(t\theta)\mathbf{q}}{\sin\theta} \tag{3.2}$$

where θ is the angle inclined by quaternions $\mathbf{p}, \mathbf{q}$, i.e., $\cos\theta = \langle\mathbf{p}, \mathbf{q}\rangle$. Note that $SLERP(t; \mathbf{p}, \mathbf{q})$ is always a unit quaternion.

The quaternion linear blending is computed as

$$QLB(t; \mathbf{p}, \mathbf{q}) = \frac{(1-t)\mathbf{p} + t\mathbf{q}}{\|(1-t)\mathbf{p} + t\mathbf{q}\|} \tag{3.3}$$

The difference to SLERP is obvious: QLB interpolates along the shortest *segment*, and then projects to arc, which does not result in the uniform interpolation of the arc. In spite of this, we claim that QLB is sufficient for our task. In order to justify this statement, we face a question interesting by itself: how big can be the difference between QLB and SLERP for the same input rotations?

Lemma 3.1. *For any unit quaternions* $\mathbf{p}$ *and* $\mathbf{q}$ *and any parameter* $t \in [0, 1]$, *we can write* $SLERP(t; \mathbf{p}, \mathbf{q}) = \mathbf{pr}_s(t)$ *and* $QLB(t; \mathbf{p}, \mathbf{q}) = \mathbf{pr}_l(t)$, *where the unit quaternions* $\mathbf{r}_s(t)$ *and* $\mathbf{r}_l(t)$ *have the same axis of rotation. Moreover, this axis is constant, i.e., independent of the interpolation parameter* t.

Proof. For $t = 0$, both QLB and SLERP return of course $\mathbf{p}$. For $t > 0$, we can imagine that both QLB and SLERP work by concatenating $\mathbf{p}$ with some rotation (multiplying $\mathbf{p}$ with some quaternion). For SLERP, we denote this quaternion as $\mathbf{r}_s(t)$. It can be expressed as $\mathbf{p}^* SLERP(t; \mathbf{p}, \mathbf{q})$, because

$$\mathbf{pr}_s(t) = \mathbf{pp}^* SLERP(t; \mathbf{p}, \mathbf{q}) = SLERP(t; \mathbf{p}, \mathbf{q})$$

The rotation $\mathbf{r}_s(t)$ can be written out as

$$\mathbf{r}_s(t) = \mathbf{p}^* SLERP(t; \mathbf{p}, \mathbf{q}) = \frac{\sin((1-t)\theta) + \sin(t\theta)\mathbf{p}^*\mathbf{q}}{\sin\theta} \tag{3.4}$$

From the definition of quaternion multiplication, it can be seen that the scalar part of $\mathbf{p}^*\mathbf{q}$ equals $\langle \mathbf{p}, \mathbf{q} \rangle = \cos\theta$. Since $\mathbf{p}^*\mathbf{q}$ is a unit quaternion, we can express it as

$$\mathbf{p}^*\mathbf{q} = \cos\theta + \mathbf{u}\sin\theta$$

for some axis of rotation $\mathbf{u} \in R^3$. If we substitute this into Formula (3.4), we obtain

$$\mathbf{r}_s(t) = \frac{\sin((1-t)\theta) + \sin(t\theta)\cos\theta}{\sin\theta} + \mathbf{u}\sin(t\theta)$$

which means that the axis of rotation given by vector $\mathbf{u}$ is independent of t (since axis does not depend on the length of its representing vector).

Let us examine the rotation $\mathbf{r}_l(t)$ following $\mathbf{p}$ in QLB:

$$\mathbf{r}_l(t) = \mathbf{p}^* QLB(t; \mathbf{p}, \mathbf{q}) = \frac{(1-t) + t\mathbf{p}^*\mathbf{q}}{\|(1-t)\mathbf{p} + t\mathbf{q}\|} = \frac{(1 - t + t\cos\theta)}{\|(1-t)\mathbf{p} + t\mathbf{q}\|} + \mathbf{u}\frac{t\sin\theta}{\|(1-t)\mathbf{p} + t\mathbf{q}\|}$$

which shows that the axis of rotation $\mathbf{r}_l(t)$ is the same as that of $\mathbf{r}_s(t)$. $\square$

From Lemma 3.1, it follows that the only difference between QLB and SLERP is in the angle of rotations $\mathbf{r}_s(t)$ and $\mathbf{r}_l(t)$. The following lemma presents an upper bound of this difference.

Lemma 3.2. *The difference in the angle of $\mathbf{r}_s(t)$ and $\mathbf{r}_l(t)$ from Lemma 3.1 is strictly less than 8.15 degrees for each $t \in [0, 1]$.*

Proof. Note that both $\mathbf{r}_s(t)$ and $\mathbf{r}_l(t)$ have a form of linear combination of quaternions 1 (zero angle rotation) and $\mathbf{p}^*\mathbf{q}$. It means that both $\mathbf{r}_s(t)$ and $\mathbf{r}_l(t)$ end up in certain 2D subspace of R^4 for each $t \in [0, 1]$. We can restrict our attention to this subspace (the linear hull of 1 and $\mathbf{p}^*\mathbf{q}$).

Since SLERP assumes $\cos\theta = \langle\mathbf{p}, \mathbf{q}\rangle \geq 0$, the angle θ cannot exceed $\pi/2$. To obtain an upper bound of the maximal difference in the angle, we consider the extremal case with $\theta = \pi/2$, depicted in Figure 3.4.
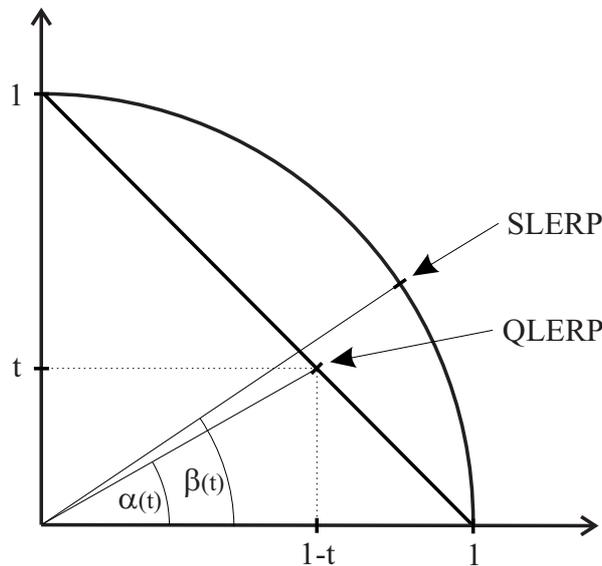
Figure 3.4: Difference between the QLB angle $\alpha(t)$ and the SLERP one, $\beta(t)$

The angle $\alpha(t)$ on the picture can be computed by *atan*, and $\beta(t)$ by simple linear interpolation of the right angle, which yields the difference function

$$d(t) = \alpha(t) - \beta(t) = atan\left(\frac{t}{1-t}\right) - \frac{\pi}{2}t$$

It remains to find the extremes of $d(t)$ on the interval $\langle 0, 1 \rangle$. An elementary mathematical analysis discovers the global extremes in points $1/2 \pm \sqrt{(1/\pi - 1/4)}$. The absolute value of $d(t)$ in these points is approximately 0.071 radians (4.07 degrees). To finish the proof, recall that the angle of rotation is twice the angle inclined by quaternions. $\square$

To conclude: both SLERP and QLB interpolate by multiplying the first quaternion with a rotation with the same, fixed axis. The difference between SLERP and QLB is only in the angle of this rotation, and is strictly less then 0.143 radians (8.15 degrees) for any interpolation parameter $t \in [0, 1]$. This is an upper bound; practical results are much smaller and could hardly cause an observable defect in the deformed skin. The big advantage of QLB over SLERP is that it can be easily generalized to interpolate multiple rotations – it suffices to make a convex combination and re-normalization of multiple quaternions (more in the next section).

3.3 Algorithm Overview

Now we have prepared all the ingredients to describe how the SBS algorithm works. The task is to transform a vertex $\mathbf{v}$ influenced by joints $j_1, \dots, j_n$ with convex weights $\mathbf{w} = (w_1, \dots, w_n)^T \in W_n$ to its position $\mathbf{v}'$ in the animated skin. In order to obtain a correct deformation, it is necessary to respect the computed center of rotation $\mathbf{r}_c$. First, however, we need to extend the QLB scheme to homogeneous matrices C_{j_i}. We denote the blending of matrices C_{j_i} with weights $\mathbf{w}$ as

$$QLB'_{\mathbf{0}}(\mathbf{w}; C_{j_1}, \dots, C_{j_n}) = \begin{pmatrix} Q & \mathbf{m} \\ \mathbf{0}^T & 1 \end{pmatrix} \qquad (3.5)$$

The subscript $\mathbf{0}$ means that the rotation is with respect to the origin (zero vector). Rotation matrix Q and translation vector $\mathbf{m}$ are computed as follows. First, the rotation submatrices $C_{j_i}^{rot}$ are converted to quaternions $\mathbf{q}_{j_i}$. One of them, for example $\mathbf{q}_{j_1}$, is chosen as pivot. If $\langle \mathbf{q}_{j_1}, \mathbf{q}_{j_i} \rangle < 0$ for any $i = 2, \dots, n$, we replace $\mathbf{q}_{j_i}$ with $-\mathbf{q}_{j_i}$ (by analogy to SLERP). Then, $QLB'_{\mathbf{0}}$ computes $\mathbf{s} = w_1\mathbf{q}_{j_1} + \dots + w_n\mathbf{q}_{j_n}$, which is subsequently normalized to $\mathbf{s}_n = \mathbf{s}/\|\mathbf{s}\|$. Finally, $\mathbf{s}_n$ is converted to the rotation matrix Q. The translation part is just linearly combined, $\mathbf{m} = \sum_{i=1}^{n} w_i \mathbf{C}_{j_i}^{tr}$.

The following lemma explains how the blending of homogeneous matrices changes when we switch the center of rotation from $\mathbf{0}$ to $\mathbf{r}_c$.

Lemma 3.3.

$$QLB'_{\mathbf{r}_c}(\mathbf{w}; C_{j_1}, \dots, C_{j_n}) = T \; QLB'_{\mathbf{0}}(\mathbf{w}; T^{-1}C_{j_1}T, \dots, T^{-1}C_{j_n}T) \; T^{-1} \qquad (3.6)$$

where T is a 4×4 matrix

$$T = \begin{pmatrix} I & \mathbf{r}_c \\ \mathbf{0}^T & 1 \end{pmatrix}$$

Proof. Let us denote by K the coordinate system with origin in $\mathbf{r}_c$ and identical basis vectors as the world coordinate system. The matrix T is nothing but transformation from K to the world coordinate system. By analogy, the inverse matrix

$$T^{-1} = \begin{pmatrix} I & -\mathbf{r}_c \\ \mathbf{0}^T & 1 \end{pmatrix}$$

represents the transformation from the world coordinate system to K. It follows that $T^{-1}C_{j_i}T$ is the transformation C_{j_i} expressed with respect to K. By blending these matrices with QLB_0', we obtain a matrix working also on vectors in K coordinates:

$$QLB_0'(\mathbf{w}; T^{-1}C_{j_1}T, \ldots, T^{-1}C_{j_n}T)$$

We can express this matrix with respect to the world coordinate system easily:

$$TQLB_0'(\mathbf{w}; T^{-1}C_{j_1}T, \ldots, T^{-1}C_{j_n}T)T^{-1}$$

which is exactly what we wanted to proof.   □

Using $QLB_{\mathbf{r}_c}'$, we can write the formula of SBS concisely: $\mathbf{v}' = QLB_{\mathbf{r}_c}'(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})\mathbf{v}$. The following lemma says how this equation can be computed more efficiently.

Lemma 3.4. *For any vertex* $\mathbf{v} \in R^3$, *the formula*

$$\mathbf{v}' = QLB_{\mathbf{r}_c}'(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})\mathbf{v}$$

simplifies to

$$\mathbf{v}' = Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i}\mathbf{r}_c \tag{3.7}$$

where $Q = QLB(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})$.

Proof. Recall that the matrix C_{j_i} has structure

$$C_{j_i} = \begin{pmatrix} C_{j_i}^{rot} & \mathbf{C}_{j_i}^{tr} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

which enables us to write out

$$T^{-1}C_{j_i}T = \begin{pmatrix} C_{j_i}^{rot} & C_{j_i}\mathbf{r}_c - \mathbf{r}_c \\ \mathbf{0}^T & 1 \end{pmatrix}$$

as can be simply verified. Therefore, according to Formula (3.5),

$$QLB_0'(\mathbf{w}; T^{-1}C_{j_1}T, \ldots, T^{-1}C_{j_n}T) = \begin{pmatrix} Q & -\mathbf{r}_c + \sum_{i=1}^{n} w_i C_{j_i}\mathbf{r}_c \\ \mathbf{0}^T & 1 \end{pmatrix}$$

where $Q = QLB(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})$. Using $T^{-1}\mathbf{v} = \mathbf{v} - \mathbf{r}_c$ and $T\mathbf{x} = \mathbf{x} + \mathbf{r}_c$, we see that

$$\begin{aligned} \mathbf{v}' &= TQLB_0'(\mathbf{w}; T^{-1}C_{j_1}T, \ldots, T^{-1}C_{j_n}T)T^{-1}\mathbf{v} \\ &= T\begin{pmatrix} Q & -\mathbf{r}_c + \sum_{i=1}^{n} w_i C_{j_i}\mathbf{r}_c \\ \mathbf{0}^T & 1 \end{pmatrix}\begin{pmatrix} \mathbf{v} - \mathbf{r}_c \\ 1 \end{pmatrix} \\ &= Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i}\mathbf{r}_c \end{aligned}$$

is true for any vector $\mathbf{v}$. Note that the shift of the center of rotation does not influence the blended rotation – it manifests only in the translation part.   □

The Formula (3.7) has to be evaluated once per each vertex. A basic optimization is to pre-compute the quaternions $\mathbf{q}_{j_i}$, because they do not depend on the actual vertex – only on the joint's transformation, similarly as the rotation centers $\mathbf{r}_c$. Nevertheless, QLB still has to be executed for each vertex, since vertex weights $w_1, \ldots, w_n$ vary. In order to challenge the speed of LBS, we apply the following trick.

According to Section 2.1.1, the rotation of vector $\mathbf{v}$ by quaternion $\mathbf{s}_n = \mathbf{s}/\|\mathbf{s}\|$ can be expressed as $\mathbf{s}_n \mathbf{v} \mathbf{s_n}^*$. Although this expression is not efficient for computation (because of slow quaternion multiplication), it enables us to write out the rotation of $\mathbf{v}$ by quaternion $\mathbf{s}_n$ as

$$\mathbf{s}_n \mathbf{v} \mathbf{s}_n{}^* = \frac{1}{\|\mathbf{s}\|^2} \mathbf{s} \mathbf{v} \mathbf{s}^* = \frac{1}{\langle \mathbf{s}, \mathbf{s} \rangle} \mathbf{s} \mathbf{v} \mathbf{s}^*$$

This suggests to convert already the quaternion $\mathbf{s}$ to matrix Q' and normalize subsequently by dividing $\langle \mathbf{s}, \mathbf{s} \rangle$. Therefore, we can compute the matrix Q from Formula (3.7) as $Q = \frac{Q'}{\langle \mathbf{s}, \mathbf{s} \rangle}$ and save the *sqrt* operation. Some attention must be paid because standard routines for quaternion to matrix conversion assume a unit-length quaternion. The conversion of an arbitrary length $\mathbf{q}' = w + xi + yj + zk$ leads to the following matrix:

$$\begin{pmatrix} x^2 + w^2 - y^2 - z^2 & 2xy - 2wz & 2xz + 2wy \\ 2xy + 2wz & y^2 + w^2 - x^2 - z^2 & 2yz - 2wx \\ 2xz - 2wy & 2yz + 2wx & z^2 + w^2 - x^2 - y^2 \end{pmatrix}$$

Vertex normal $\mathbf{v}_n$ is transformed in a similar way as vertex position, but ignoring the translation component

$$\mathbf{v}'_n = Q\mathbf{v}_n$$

Using Formula (3.7), we can verify our previous intuitive reasoning. If $n = 2$ and $C_{j_1}\mathbf{r}_c = C_{j_2}\mathbf{r}_c$ (as in the beginning of Section 3.1), the translation part becomes

$$w_1 C_{j_1}\mathbf{r}_c + w_2 C_{j_2}\mathbf{r}_c = (w_1 + w_2)C_{j_1}\mathbf{r}_c = C_{j_1}\mathbf{r}_c = C_{j_2}\mathbf{r}_c$$

which is independent of blending parameters (weights), i.e., no translation is blended, as in Figure 3.3. The whole SBS is summarized in Algorithm 3.1. The algorithm uses a subroutine ROTCENTER, which computes the center of rotation using SVD as discussed in Section 3.1.

3.4   Results and Comparison

We tested the SBS algorithm on various models (see Figure 3.5 and Table 3.1) and compared the shape of the deformed skin on the model of woman, because human eye is most sensitive to the deformations of human body. Figure 3.6 presents the results of LBS and SBS executed on the same posture of the model. Another example has been presented in Figure 3.1. For small deformations, both algorithms produce similar results, as in the second column of Figure 3.6 (although a small loss of volume is noticeable even there).

Algorithm 3.1: Spherical blend skinning

Input: $C_1, \ldots, C_p$ – joint transformation matrices for the current posture

$\mathbf{v}_1, \ldots, \mathbf{v}_m$ – rest-pose vertices

$\nu_1, \ldots, \nu_m$ – rest-pose normals

$w_{1,1}, \ldots, w_{1,n_1}, \ldots, w_{m,1}, \ldots, w_{m,n_m}$ – vertex weights

$j_{1,1}, \ldots, j_{1,n_1}, \ldots, j_{m,1}, \ldots, j_{m,n_m}$ – influencing joints indices

(see Section 2.2.1 for description)

Output: $\mathbf{v}'_1, \ldots, \mathbf{v}'_m$ – vertices of the deformed skin

$\nu'_1, \ldots, \nu'_m$ – normals of the deformed skin

$\text{SBS}(C_1, \ldots, C_p)$

(1)   **for** $k = 1$ **to** m

(2)   **if** rotation center for $j_{k,1}, \ldots, j_{k,n_k}$ not in cache

(3) $\mathbf{r}_c = \text{ROTCENTER}(C_{j_{k,1}}, \ldots, C_{j_{k,n_k}})$

(4)   store $\mathbf{r}_c$ in cache for key $j_{k,1}, \ldots, j_{k,n_k}$

(5)   **else**

(6)   retrieve $\mathbf{r}_c$ from cache for key $j_{k,1}, \ldots, j_{k,n_k}$

(7) $Q = QLB(w_{k,1}, \ldots, w_{k,n_k}; C_{j_{k,1}}, \ldots, C_{j_{k,n_k}})$

(8) $\mathbf{v}'_k = Q(\mathbf{v}_k - \mathbf{r}_c) + \sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}} \mathbf{r}_c$

(9) $\nu'_k = Q \nu_k$

Figure 3.5: 3D models used for testing

It is remarkable that the results of SBS look better even though the models have been optimized to work with the LBS algorithm.

The performance of both algorithms is compared in Table 3.2. The measured value is an average time in milliseconds necessary to deform one model on a 3.4GHz Pentium 4 (rendering time not included). We see that the time complexity of SBS, though higher than

	Hand	Woman	Creature
vertices	2402	3356	6802
triangles	4800	5205	13590
joints	23	78	56

Table 3.1: Complexities of example models (testing of spherical blend skinning)
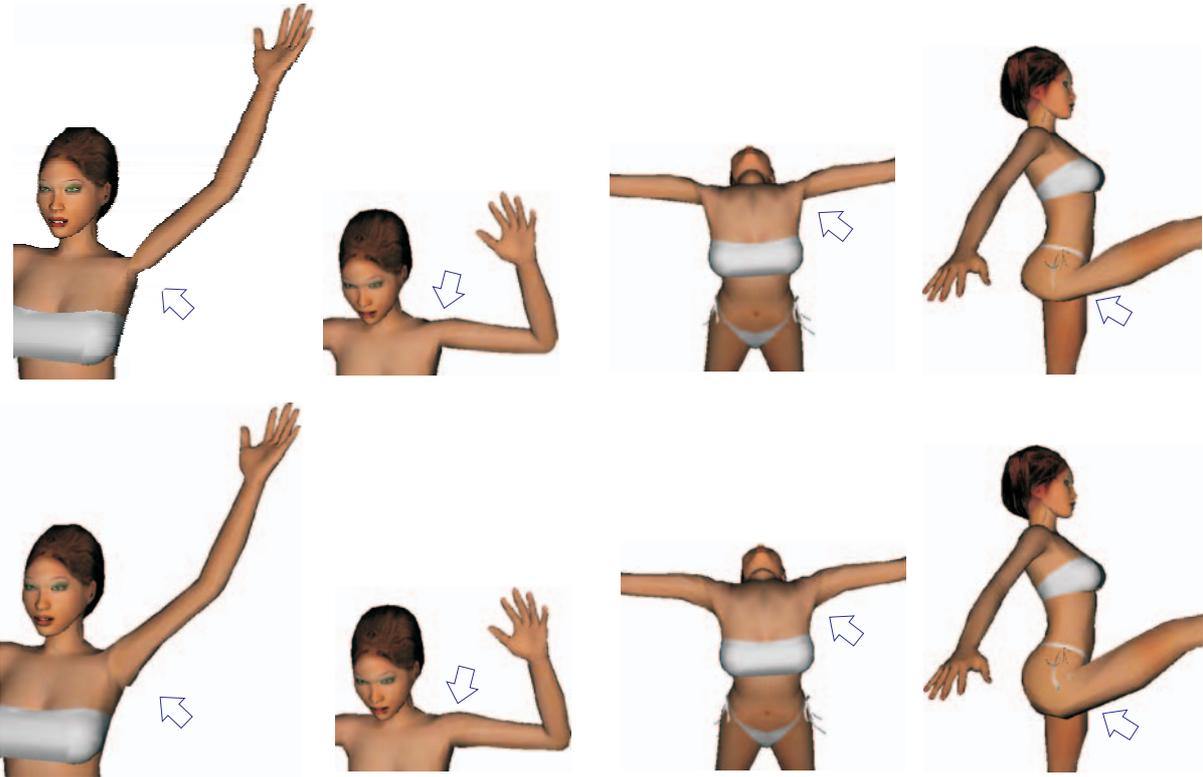
Figure 3.6: Comparison of the deformations computed by linear (top) and spherical blend skinning (bottom)

that of LBS, is still quite suitable for real-time applications. In the last row of the table the number of different non-trivial joint sets is reported. Put in another way, it is exactly the number of singular value decompositions performed by the SBS algorithm. Theoretically, the number of different non-trivial joint sets could be very high. Fortunately, this number is rather small in practice, because the joint influences tend to be local (e.g., it is very unlikely to find vertices influenced by both left and right wrist).

The additional memory needed for SBS is dominated by caching the computed centers of rotation. However, this amount of memory is negligible, considering the number of different non-trivial joint sets.

We also tried to apply the spherical weighted averages [22] instead of QLB. Spherical averages behave like SLERP for the case of two rotations, in contrast to QLB, which only

	Hand	Woman	Creature
LBS time	2.68	2.81	7.6
SBS time	5.79	5.43	16.11
SVD executions	38	37	56

Table 3.2: First two rows: run-time of LBS and SBS algorithms in milliseconds; last row: number of SVD executions

approximates the SLERP results. On one hand, the difference in the deformed skin was barely observable (confirming the results from Section 3.2). On the other hand, the increase in the execution time was quite substantial (spherical averages took about 5 times longer). This experiment confirmed our choice of QLB for rotation blending.

4 Dual Quaternion Skinning

The spherical blend skinning algorithm, introduced in Chapter 3, achieves our goal of removing the artifacts of linear blend skinning while retaining its real-time efficiency. However, it is obvious that the solution by spherical blending lacks the simplicity and elegancy of linear blending. Spherical blending requires the non-trivial singular value decomposition algorithm as a subroutine. Also, the caching of rotation centers introduces potential problems when moving the algorithm from CPU to GPU. This chapter therefore analyzes the properties of spherical blend skinning, in order to find a skinning algorithm similar in quality, but faster to execute and amenable to graphics hardware. We demonstrate that such an algorithm can be obtained using dual quaternions.

4.1   Properties of Spherical Blending

The main problem of SBS is the computation of the center of rotation – this is also the part which requires the singular value decomposition. Moreover, as rotation centers are shared by multiple vertices, there is a potential source of discontinuities in parts of the skin where the rotation centers change. Even though we usually cannot observe this problem in practice, it is possible to design an example where this produces visual artifacts, see Figure 4.1.

Ideally, we would like a skinning algorithm similar to spherical blending, but avoiding the computation of the center of rotation. First, we have to clarify what we mean by "similar to spherical blending". As we have seen in the examples, SBS produces reasonable skin deformations, while the direct application of QLB_0' does not. Nevertheless, spherical blending is nothing but QLB_0' with a different center of rotation. Can we explain mathematically why one blending algorithm does not work, whereas a very similar one does? Let us denote a general blending method among z rigid transformations $M_1, \ldots, M_z$ with weight vector

Rest-pose   Spherical Blend Skinning   Dual Quaternions

Figure 4.1: Piece of cloth deformed by spherical blend skinning reveals artifacts caused by discontinuous change of rotation centers. Dual quaternion skinning (Section 4.5) does not have such problems and produces a smooth deformation.

$\mathbf{w} \in W_z$ as $\Phi(\mathbf{w}; M_1, \ldots, M_z)$. For brevity, let us further denote the spherical blending as Φ_{SB}. We might now ask which property that Φ_{SB} fulfills does not hold for QLB'_0.

A general blending method Φ is said to be *left-invariant*, if

$$\forall U \in SE(3) : U\Phi(\mathbf{w}; M_1, \ldots, M_z) = \Phi(\mathbf{w}; UM_1, \ldots, UM_z)$$

By analogy, we define *right-invariance* of Φ as

$$\forall U \in SE(3) : \Phi(\mathbf{w}; M_1, \ldots, M_z)U = \Phi(\mathbf{w}; M_1 U, \ldots, M_z U)$$

(Recall that SE(3) is the group of rigid transformations, see Section 2.1.) If both left and right-invariance applies at the same time, we speak about *bi-invariance* [98]. Bi-invariance directly implies another important property: *coordinate-invariance*, which is defined as

$$\forall U \in SE(3) : U\Phi(\mathbf{w}; M_1, \ldots, M_z)U^{-1} = \Phi(\mathbf{w}; UM_1 U^{-1}, \ldots, UM_z U^{-1})$$

The fact that bi-invariance implies coordinate-invariance is obvious. We will see below that bi-invariance is a stronger property (i.e., that bi-invariance is not equivalent to coordinate invariance).

Let us clarify the geometrical interpretation of left/right-invariance and their implications to skinning. Left-invariance considers the input transformations M_i multiplied from the left by U, i.e., M_i is applied first and U operates on the result. This means that in the case of left-invariance, U acts in world-space coordinates. In skinning, left-invariance has the following interpretation: the matrices $UM_1, \ldots, UM_z$ mean simply transformation of the whole posture in the world space. Left-invariance therefore expresses the fact that the skinning of a transformed model is equivalent to the transformation of a skinned model. This a very natural requirement; if breached, unpleasant artifacts can occur. It is easy to see that QLB'_0 is left-invariant, as well as spherical blending. The latter is proven in the following lemma. First, let us remark that QLB (i.e., rotation-only blending) is both left and right invariant, which follows from the left and right distributivity of quaternion multiplication (see a more general Lemma 4.5 in Section 4.3).

Lemma 4.1. *If Φ_{SB} denotes spherical blending (Formula (3.7)), then*

$$\forall U \in SE(3) : U\Phi_{SB}(\mathbf{w}; C_{j_1}, \ldots, C_{j_n}) = \Phi_{SB}(\mathbf{w}; UC_{j_1}, \ldots, UC_{j_n})$$

Proof. Let us assume that the rotation center computed for matrices $C_{j_1}, \ldots, C_{j_n}$ is $\mathbf{r}_c$. What will be the rotation center computed for matrices $UC_{j_1}, \ldots, UC_{j_n}$? Substituting into Formula (3.1) yields

$$UC_a\mathbf{x} = UC_b\mathbf{x}, \; a < b, \;\; a, b \in \{j_1, \ldots, j_n\}$$

Multiplying from the left by U^{-1}, we see that this is equivalent to the original system and therefore $\mathbf{x} = \mathbf{r}_c$.

Let us denote the components of matrix U as follows:

$$U = \begin{pmatrix} U^{rot} & \mathbf{U}^{tr} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

Left-multiplication of the SBS formula by U therefore gives

$$U \left(Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c \right) = U^{rot} \left(Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c \right) + U^{tr}$$

On the other hand, SBS for transformations $UC_{j_1}, \ldots, UC_{j_n}$ yields, applying the left-invariance of QLB,

$$
\begin{aligned}
U^{rot} Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i U C_{j_i} \mathbf{r}_c &=& U^{rot} Q(\mathbf{v} - \mathbf{r}_c) + U \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c = \\
&=& U^{rot} \left(Q(\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c \right) + U^{tr}
\end{aligned}
$$

By comparing these equations we can conclude that spherical blending is indeed left-invariant. $\square$

Let us now turn our attention to the right-invariance and its geometrical interpretation. Right-invariance considers the input transformations M_i multiplied from right by U, i.e., M_i operates on a point already transformed by U. This means that in the case of right-invariance, U acts in body-space coordinates. In skinning, this expresses the following fact: if the body-space coordinates of model vertices $\mathbf{v}_1, \ldots, \mathbf{v}_m$ are changed to $U\mathbf{v}_1, \ldots, U\mathbf{v}_m$, then this will be compensated by changing matrices $C_1, \ldots, C_p$ to $C_1 U^{-1}, \ldots, C_p U^{-1}$ (this happens, for example, when a designer decides to translate the model's origin). The right-invariance then requires that this does not influence the skin deformation, i.e., that the result will be the same as without the translation U.

Obviously, $QLB'_{\mathbf{0}}$ cannot be right-invariant, because this would already imply coordinate-invariance. This would mean that Lemma 3.3 could have been simplified to

$$QLB'_{\mathbf{r}_c}(\mathbf{w}; C_{j_1}, \ldots, C_{j_n}) = QLB'_{\mathbf{0}}(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})$$

which is obviously not true. In other words, right-variance means that a transformed object rotates with respect to the origin of its body-space coordinates, i.e., a user-specified point (see Figure 3.3). With a right-invariant blending, the body-space coordinates do not determine the rotation center anymore. The rotation center is thus given by other means, for example as the point yielding a minimal drift of blended transformations (see Section 3.1). The following lemma verifies that spherical blending is right invariant.

Lemma 4.2. *If* Φ_{SB} *denotes spherical blending (Formula (3.7)), then*

$$\forall U \in SE(3) : \Phi_{SB}(\mathbf{w}; C_{j_1}, \ldots, C_{j_n})U = \Phi_{SB}(\mathbf{w}; C_{j_1}U, \ldots, C_{j_n}U)$$

Proof. Let us denote the rotation center computed for matrices $C_{j_1}, \ldots, C_{j_n}$ as $\mathbf{r}_c$. What will be the rotation center $\mathbf{r}_c'$ for matrices $C_{j_1}U, \ldots, C_{j_n}U$? Formula 3.1 expresses the rotation center as the least squares solution of

$$C_a U \mathbf{x} = C_b U \mathbf{x}, \ a < b, \ a, b \in \{j_1, \ldots, j_n\}$$

Let us introduce the substitution $\mathbf{y} = U\mathbf{x}$, which converts the system to the original form $C_a \mathbf{y} = C_b \mathbf{y}$. The solution therefore is $\mathbf{y} = \mathbf{r}_c$ (i.e., the original rotation center) and $\mathbf{x} = U^{-1}\mathbf{r}_c$.

Let us denote the components of matrix U as follows:

$$U = \begin{pmatrix} U^{rot} & \mathbf{U}^{tr} \\ \mathbf{0}^T & 1 \end{pmatrix}$$

If we apply Formula (3.7) to $U\mathbf{v}$, we obtain

$$Q(U\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c$$

When we apply SBS to matrices $C_{j_1}U, \ldots, C_{j_n}U$ and vertex $\mathbf{v}$, we obtain, using the right-invariance of QLB,

$$QU^{rot}(\mathbf{v} - U^{-1}\mathbf{r}_c) + \sum_{i=1}^{n} w_i (C_{j_i}U)U^{-1}\mathbf{r}_c =$$

$$= QU^{rot}(\mathbf{v} - (U^{rot})^T \mathbf{r}_c + (U^{rot})^T U^{tr}) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c =$$

$$= QU^{rot}\mathbf{v} - Q\mathbf{r}_c + QU^{tr} + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c = Q(U\mathbf{v} - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c$$

This shows the right-invariance of spherical blending. □

Therefore, right-invariance is the key property for correct skinning, because QLB_0' is equivalent to spherical blending up to right-invariance. Unfortunately, the right-invariance of spherical blending is achieved by computing the rotation center, which has the disadvantages discussed in the beginning of this section. Obviously, it would be very desirable to design a right-invariant rigid transformation blending which avoids the computation of the rotation center. Of course, such a blending method must not breach other desirable properties of QLB_0', such as left-invariance and shortest-path (see Section 3.1).

A promising idea is to base the potential advanced blending algorithm on screw parameters of a rigid transformation (see Section 2.1.2). A straightforward way would be to convert the skinning matrices $C_{j_1}, \ldots, C_{j_n}$ to screw parameters (axis $\mathbf{a}$, moment $\mathbf{m}$, twist θ and pitch d) and blend them linearly with given weights. Unfortunately, this method fails in
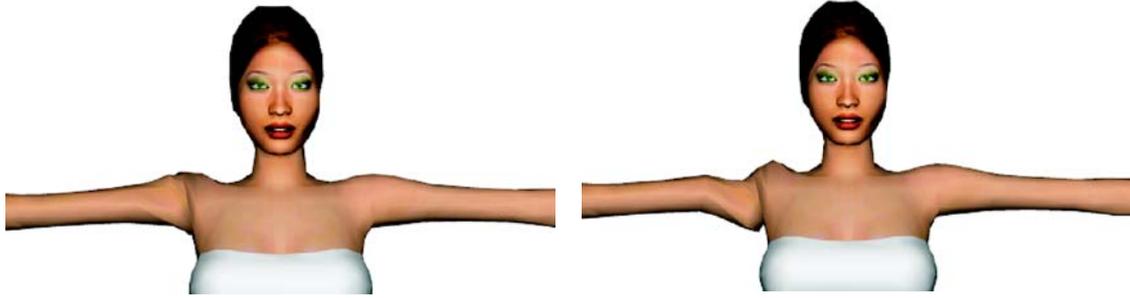
Figure 4.2: Left: In most positions, log-matrix blending solves the candy wrapper artifact equally well as spherical blending. Right: However, translation of the model in the world space results in artifacts, due to the left-variance of log-matrix blending.

the same way as QLB_0': linear blending of screw parameters is not right-invariant. This is a consequence of the fact that the multiplication of 4-tuples $(\mathbf{a}, \mathbf{m}, \theta, d)$ defined via screw composition is not distributive.

Alexa proposed another interesting method: linear blending of matrix logarithms, followed by matrix exponential [3]. This method is indeed coordinate-invariant, because matrix logarithm and exponential satisfy the relations

$$\exp(TMT^{-1}) = T\exp(M)T^{-1}, \quad \log(TMT^{-1}) = T\log(M)T^{-1},$$

see for example [98]. This is what enables the application of Alexa's method in skinning without computing rotation centers [28].

Alexa's log-matrix blending produces reasonable skin deformations in most situations. Unfortunately, matrix logarithms do not blend well the rotation parts of input matrices [13] (see also Section A.2 in the Appendix). Another problem of log-matrix blending is that it is neither left, nor right-invariant (which also shows that bi-invariance is really a stronger property than coordinate-invariance). This follows from the simple fact that $\exp(TM) \neq T\exp(M)$ as well as $\exp(MT) \neq \exp(M)T$ (and analogically for logarithms). To see this, it is sufficient to consider just the case when M is a scalar, i.e., a 1×1 matrix. The consequence of the left-variance of log-matrix blending is illustrated in Figure 4.2 (right-variance produces similar artifacts). This is undesirable, even though in most practical situations the artifacts are not as disturbing as in Figure 4.2.

For completeness, let us examine the invariance properties of linear blending. Linear blending of matrices is bi-invariant (and thus also automatically coordinate-invariant), because for any matrix T, the following is true:

$$\left(\sum_{i=1}^{n} w_i T M_i\right) = T\left(\sum_{i=1}^{n} w_i M_i\right), \quad \left(\sum_{i=1}^{n} w_i M_i T\right) = \left(\sum_{i=1}^{n} w_i M_i\right) T$$

This is a consequence of the distributivity of matrix products. The bi-invariance of linear blend skinning explains why linear blending has no problems with any change of coordinates. This is probably one of the reasons why linear blend skinning still dominates in many applications, in spite of the skin collapsing artifacts.

To conclude, we see that none of the previous solutions works perfectly in skinning: a novel rigid transformation blending algorithm would be desirable. We found interesting the intimate relationships between algebra, geometry and computer animation: distributive property of multiplications reflects itself in coordinate invariance, which in turn has implications to skinning.

4.2 Optimal Rigid Transformation Blending for Skinning

In this section, we summarize (and formalize) the properties of an ideal rigid transformation blending algorithm. We discuss this problem in general, because rigid transformation blending has potentially much more applications than just skinning, e.g., motion blending and animation compression [3].

Any blending technique of z rigid transformations $M_1, \ldots, M_z$ with weight vector $\mathbf{w} \in W_z$, denoted as $\Phi(\mathbf{w}; M_1, \ldots, M_z)$, must satisfy the following basic properties:

1. $\forall i = 1, \ldots, z : \Phi(\mathbf{e}_i; M_1, \ldots, M_z) = M_i$

2. $\forall \mathbf{w} \in W_z : \Phi(\mathbf{w}; M_1, \ldots, M_z) \in SE(3)$

3. $\forall$ permutation π of $\{1, \ldots, z\}$:

$$\Phi((w_{\pi(1)}, \ldots, w_{\pi(z)})^T; M_{\pi(1)}, \ldots, M_{\pi(z)}) = \Phi(\mathbf{w}; M_1, \ldots, M_z)$$

The first condition expresses the fact that if all weights except one are zero, then the result is exactly the transformation corresponding to the non-zero element (recall that $\mathbf{e}_i$ is the i-th standard basis vector – see Section 2.1). The second condition requires that the blending method preserves rigidity, i.e., enables us to speak about *rigid transformation blending.* The third condition simply means that the blending is not dependent on any permutation of the input. This condition may sound obvious, but consider that, for example, that successive SLERPs do not satisfy permutation invariance [22].

However, even when taking all these requirements into account, there is still an infinite number of ways how to define a valid blending. It is thus desirable to pick a blending method which satisfies the most advantageous additional properties. One of such properties, discussed in Section 4.1, is bi-invariance, i.e.,

$$\forall T \in SE(3): \quad T\Phi(\mathbf{w}; M_1, \ldots, M_z) = \Phi(\mathbf{v}; TM_1, \ldots, TM_z)$$
$$\Phi(\mathbf{w}; M_1, \ldots, M_z)T = \Phi(\mathbf{v}; M_1 T, \ldots, M_z T)$$

Other two useful properties can be taken from the well-known SLERP algorithm [117], which interpolates between two 3D rotations (R_1, R_2). These properties are:

- *Constant speed*, meaning that the angle of interpolated rotation varies linearly with respect to the interpolation parameter.

- *Shortest path*, meaning that the motion between R_1 and R_2 is a rotation about a fixed axis with the smallest angle.

It is relatively simple to generalize these properties to our case of rigid transformations. Let us decompose the rigid transformation $M_1^{-1}\Phi(t; M_1, M_2), t \in [0, 1]$ to corresponding screw parameters: axis $\mathbf{a}(t)$, moment $\mathbf{m}(t)$, twist $\theta(t)$ and pitch $d(t)$. Then a rigid transformation blending is said to be

- *Constant speed* if the derivative of both $\theta(t)$ and $d(t)$ is constant.

- *Shortest path* if $\mathbf{a}(t)$ and $\mathbf{m}(t)$ are constant, and $\theta(1) \in [-\pi, \pi]$.

Note that even though the constant speed and shortest path properties consider only the case of two rigid transformations (i.e., interpolation), they can still be used with a general blending (by considering the remaining weights to be zero). More formally, we could have defined the constant speed and shortest path in general by requiring constant speed and shortest path interpolation for each set of weights $w_1(t), \ldots, w_n(t)$ corresponding to an R^n line segment for $t \in [0, 1]$. However, both definitions are equivalent in the case of blending defined via linear combinations. Since this kind of blending is our main concern in this thesis, we prefer to use the definition stated above.

4.3   Dual Quaternion Blending

In this section, we show how dual quaternions can be exploited to design a rigid transformation blending algorithm satisfying the properties from Section 4.2. One might wonder whether it is really necessary to apply (and therefore study) dual quaternions, because any algorithm working with dual quaternions can be re-written to an algorithm working just with regular quaternions (which, in turn, can be re-written just to basic linear algebra). In fact, our first attempt was to derive blending algorithms with the desired properties only using classical quaternions, but the resulting formulas quickly became too complicated. We hypothesize that this process would not result in anything but re-discovering of dual quaternions (which seems to be, however, a problem far beyond the scope of this thesis).

Using dual quaternions, we design three practical rigid transformation blending algorithms. They are derived by generalizing previous successful rotation blending techniques to all rigid transformations:

- **ScLERP** (Screw Linear Interpolation), a generalization of SLERP [117]

- **DLB** (Dual quaternion Linear Blending), a generalization of QLB (see Section 3.2)

- **DIB** (Dual quaternion Iterative Blending), a generalization of spherical averages [22]

ScLERP fulfills all properties from Section 4.2, but works only for two rigid transformations (therefore we call it interpolation rather then blending). DLB can be applied to more than two rigid transformations, but is only approximately constant-speed (we provide upper bounds of the error). Finally, DIB meets all properties from Section 4.2 and supports more than two rigid transformations. However, DIB is iterative, and therefore can be slower to compute than the closed-form DLB. Our comparison with previous rigid transformation blending algorithms (Section 4.4) shows that dual quaternion methods are in most cases more favorable, both in terms of mathematical properties and computational speed.

We start by generalizing the famous Spherical Linear Interpolation (SLERP) from rotations to rigid transformations. The interpolation of two unit quaternions $\mathbf{p}, \mathbf{q}$ with parameter $t \in [0, 1]$ is computed as $SLERP(t; \mathbf{p}, \mathbf{q}) = \mathbf{p}(\mathbf{p}^*\mathbf{q})^t$ (assuming that we have already enforced $\langle \mathbf{p}, \mathbf{q} \rangle \geq 0$). In Section 2.1.2, we have defined the power of dual quaternions, which enables us to straightforwardly generalize SLERP to Screw Linear Interpolation (ScLERP). ScLERP interpolates between two unit dual quaternions $\hat{\mathbf{p}}, \hat{\mathbf{q}}$ with parameter $t \in [0, 1]$, and is given as $ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}}) = \hat{\mathbf{p}}(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t$. What is its geometric interpretation? Obviously, $\hat{\mathbf{p}}^*\hat{\mathbf{q}}$ is a unit dual quaternion, which represents a rigid transformation from $\hat{\mathbf{p}}$ to $\hat{\mathbf{q}}$. According to Section 2.1.2, the power can be written as $(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t = \cos(t\frac{\hat{\alpha}}{2}) + \hat{\mathbf{n}}\sin(t\frac{\hat{\alpha}}{2})$ for some dual angle $\hat{\alpha}$ and dual vector $\hat{\mathbf{n}}$. The dual vector $\hat{\mathbf{n}}$ represents the axis of the screw motion (an axis that needs not pass through the origin, as in Figure 2.3). The dual angle $t\frac{\hat{\alpha}}{2} = t\frac{\alpha_0}{2} + \epsilon t\frac{\alpha_\epsilon}{2}$ contains both the angle of rotation ($t\frac{\alpha_0}{2}$) and the amount of translation ($t\frac{\alpha_\epsilon}{2}$). We can immediately observe two important properties: the axis $\hat{\mathbf{n}}$ of the screw motion is constant (independent of t), and the angle of rotation $t\frac{\alpha_0}{2}$, as well as the amount of translation $t\frac{\alpha_\epsilon}{2}$, vary linearly with respect to the interpolation parameter t. This means that ScLERP guarantees both shortest path and constant speed interpolation. The bi-invariance of ScLERP is proven in the following two lemmas.

Lemma 4.3. *Let $\hat{\mathbf{q}} = \cos\frac{\hat{\theta}}{2} + \hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2}$, where $\hat{\mathbf{q}}, \hat{\mathbf{s}}$ are unit dual quaternions, and $\hat{\mathbf{s}}$ has zero scalar part. Then for any unit dual quaternion $\hat{\mathbf{m}}$, both of the following equations are true:*

$$\exp\left(\hat{\mathbf{m}}\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\hat{\mathbf{m}}^*\right) = \hat{\mathbf{m}}\exp\left(\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\right)\hat{\mathbf{m}}^*$$
$$\log\left(\hat{\mathbf{m}}\hat{\mathbf{q}}\hat{\mathbf{m}}^*\right) = \hat{\mathbf{m}}\log(\hat{\mathbf{q}})\hat{\mathbf{m}}^*$$

Proof. Since the scalar part of $\hat{\mathbf{s}}$ is zero, the same is true for the scalar part of $\hat{\mathbf{m}}\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\hat{\mathbf{m}}^*$, as can be shown by direct computation. This means that the exp on the left hand side of the

first equation is well defined and, according to its definition,

$$\exp\left(\hat{\mathbf{m}}\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\hat{\mathbf{m}}^*\right) = \cos\frac{\hat{\theta}}{2} + \hat{\mathbf{m}}\hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2}\hat{\mathbf{m}}^* = \hat{\mathbf{m}}\left(\cos\frac{\hat{\theta}}{2} + \hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2}\right)\hat{\mathbf{m}}^*$$

because a dual number always commutes with a dual quaternion and $\hat{\mathbf{m}}\hat{\mathbf{m}}^* = 1$. This shows the first equation. The proof of the second one is similar:

$$\hat{\mathbf{m}}\hat{\mathbf{q}}\hat{\mathbf{m}}^* = \hat{\mathbf{m}}\left(\cos\frac{\hat{\theta}}{2} + \hat{\mathbf{s}}\sin\frac{\hat{\theta}}{2}\right)\hat{\mathbf{m}}^* = \cos\frac{\hat{\theta}}{2} + \hat{\mathbf{m}}\hat{\mathbf{s}}\hat{\mathbf{m}}^*\sin\frac{\hat{\theta}}{2}$$

Therefore $\log(\hat{\mathbf{m}}\hat{\mathbf{q}}\hat{\mathbf{m}}^*) = \hat{\mathbf{m}}\hat{\mathbf{s}}\frac{\hat{\theta}}{2}\hat{\mathbf{m}}^* = \hat{\mathbf{m}}\log(\hat{\mathbf{q}})\hat{\mathbf{m}}^*$, as we wanted to prove. □

Lemma 4.4. *ScLERP is bi-invariant, that is for any unit dual quaternions $\hat{\mathbf{r}}, \hat{\mathbf{p}}, \hat{\mathbf{q}}$ and any interpolation parameter $t \in [0, 1]$, both of the following equations are true:*

$$ScLERP(t; \hat{\mathbf{r}}\hat{\mathbf{p}}, \hat{\mathbf{r}}\hat{\mathbf{q}}) = \hat{\mathbf{r}}ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$$
$$ScLERP(t; \hat{\mathbf{p}}\hat{\mathbf{r}}, \hat{\mathbf{q}}\hat{\mathbf{r}}) = ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})\hat{\mathbf{r}}$$

Proof. The left-invariance is easy, because

$$ScLERP(t; \hat{\mathbf{r}}\hat{\mathbf{p}}, \hat{\mathbf{r}}\hat{\mathbf{q}}) = \hat{\mathbf{r}}\hat{\mathbf{p}}(\hat{\mathbf{p}}^*\hat{\mathbf{r}}^*\hat{\mathbf{r}}\hat{\mathbf{q}})^t = \hat{\mathbf{r}}\hat{\mathbf{p}}(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t = \hat{\mathbf{r}}ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$$

Proving the right-invariance is a little more tricky: $ScLERP(t; \hat{\mathbf{p}}\hat{\mathbf{r}}, \hat{\mathbf{q}}\hat{\mathbf{r}}) = \hat{\mathbf{p}}\hat{\mathbf{r}}(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}})^t$. It is now sufficient to show that $(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}})^t = \hat{\mathbf{r}}^*(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t\hat{\mathbf{r}}$, because this gives us $\hat{\mathbf{p}}\hat{\mathbf{r}}(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}})^t = \hat{\mathbf{p}}\hat{\mathbf{r}}\hat{\mathbf{r}}^*(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t\hat{\mathbf{r}} = ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})\hat{\mathbf{r}}$. However, the power can be written as $(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}})^t = \exp(t\log(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}}))$. Thanks to Lemma 4.3, we can derive

$$\exp(t\log(\hat{\mathbf{r}}^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}\hat{\mathbf{r}})) = \exp(t\hat{\mathbf{r}}^*\log(\hat{\mathbf{p}}^*\hat{\mathbf{q}})\hat{\mathbf{r}}) = \hat{\mathbf{r}}^*\exp(t\log(\hat{\mathbf{p}}^*\hat{\mathbf{q}}))\hat{\mathbf{r}} = \hat{\mathbf{r}}^*(\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t\hat{\mathbf{r}}$$

which concludes the proof. □

Naturally, the bi-invariance of ScLERP implies its coordinate-invariance. The remaining properties from Section 4.2 are easy to show for ScLERP (permutation invariance follows from Lemma 4.6).

Let us now turn our attention to quaternion linear blending (QLB), introduced in Section 3.2. In the case of two transformations, the inputs of QLB are identical to those of SLERP: quaternions $\mathbf{p}, \mathbf{q}$ and parameter $t \in [0, 1]$. The disadvantage of QLB is that it is not a constant speed interpolation, although it is shortest path (Section 3.2), and coordinate-invariant, as proven below (for the more general case of dual quaternions). As computed in Section 3.2, the difference between the angle of rotation in QLB and SLERP is fairly small, i.e., always strictly less than 8.15 degrees. This means that QLB is actually "almost" constant speed interpolation, but faster and easier to compute than SLERP.

The dual counterpart of QLB is Dual quaternion Linear Blending (DLB). As could be expected, this interpolation is again a direct generalization, defined as

$$DLB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}}) = \frac{(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}}{\|(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}\|}$$

The natural question is whether DLB is also a good approximation of ScLERP, as QLB was of SLERP. We show first the bi-invariance of DLB.

Lemma 4.5. *For any unit dual quaternions $\hat{\mathbf{r}}, \hat{\mathbf{p}}, \hat{\mathbf{q}}$ and any interpolation parameter $t \in [0, 1]$, both of the following equations are true:*

$$DLB(t; \hat{\mathbf{r}}\hat{\mathbf{p}}, \hat{\mathbf{r}}\hat{\mathbf{q}}) = \hat{\mathbf{r}}DLB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$$
$$DLB(t; \hat{\mathbf{p}}\hat{\mathbf{r}}, \hat{\mathbf{q}}\hat{\mathbf{r}}) = DLB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})\hat{\mathbf{r}}$$

Proof. For proof of the left-invariance it is sufficient to use the left-distributivity of dual quaternions which implies $\|(1-t)\hat{\mathbf{r}}\hat{\mathbf{p}} + t\hat{\mathbf{r}}\hat{\mathbf{q}}\| = \|\hat{\mathbf{r}}\|\|(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}\| = \|(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}\|$ (recall that $\hat{\mathbf{r}}$ is a unit dual quaternion). We can thus write

$$DLB(t; \hat{\mathbf{r}}\hat{\mathbf{p}}, \hat{\mathbf{r}}\hat{\mathbf{q}}) = \frac{(1-t)\hat{\mathbf{r}}\hat{\mathbf{p}} + t\hat{\mathbf{r}}\hat{\mathbf{q}}}{\|(1-t)\hat{\mathbf{r}}\hat{\mathbf{p}} + t\hat{\mathbf{r}}\hat{\mathbf{q}}\|} = \hat{\mathbf{r}}\frac{(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}}{\|(1-t)\hat{\mathbf{p}} + t\hat{\mathbf{q}}\|} = \hat{\mathbf{r}}DLB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$$

Proof of the right-invariance is a direct analogy of the proof above (using right-distributivity of dual quaternions). $\square$

The left-invariance of both DLB and ScLERP simplifies their comparison. The following derivation is in fact an extension of the results from Section 3.2, and therefore we will now proceed more quickly. Instead of comparing $DLB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$ directly with $ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$, we rewrite them as $\hat{\mathbf{p}}DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$ and $\hat{\mathbf{p}}ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$, which is correct because of left-invariance. Since $\hat{\mathbf{p}}$ is the same in both expressions, it is sufficient to compare just $DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$ with $ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$, which is an easier problem. As $\hat{\mathbf{p}}^*\hat{\mathbf{q}}$ is a unit dual quaternion, it can be written as $\hat{\mathbf{p}}^*\hat{\mathbf{q}} = \cos\frac{\hat{\alpha}}{2} + \hat{\mathbf{n}}\sin\frac{\hat{\alpha}}{2}$. This enables us to derive

$$DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}}) = \frac{1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|} = \frac{1 - t + t\cos(\frac{\hat{\alpha}}{2}) + \hat{\mathbf{n}}t\sin(\frac{\hat{\alpha}}{2})}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|}$$
$$ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}}) = 1(1^*\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t = (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t = \cos\left(t\frac{\hat{\alpha}}{2}\right) + \hat{\mathbf{n}}\sin\left(t\frac{\hat{\alpha}}{2}\right)$$

from which we see that both DLB and ScLERP use the same, constant screw axis $\hat{\mathbf{n}}$. This means that DLB is a shortest path interpolation. Thus, the only difference between DLB and ScLERP is in the angle of rotation and amount of translation. Since $DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$ is a unit dual quaternion, it can be written in form

$$DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}}) = \cos\frac{\hat{\beta}_t}{2} + \hat{\mathbf{n}}\sin\frac{\hat{\beta}_t}{2}$$

By considering only the scalar part of this equality, we see that

$$\cos\frac{\hat{\beta}_t}{2} = \frac{1 - t + t\cos(\frac{\hat{\alpha}}{2})}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|}$$

In order to compute an upper bound of the difference between DLB and ScLERP, it is sufficient to express the dual angle $\hat{\beta}_t$ and compare it with ScLERP's $\hat{\alpha}t$. It is not difficult but is a lengthy mathematical analysis: we therefore employed the software package Maple [24] to carry out the computations, see Section A.1 in the Appendix. The result is that the angles of rotation in DLB and ScLERP always differ by less than 8.15 degrees (which is in accordance with the results from Section 3.2). The amount of translation always differs by less than 15.1% of the translation present in $\hat{\mathbf{p}}^*\hat{\mathbf{q}}$. Note that those results are *upper* bounds – in practice, the difference is usually much smaller. To conclude, DLB is coordinate invariant, shortest path and "almost" constant speed. DLB works also for multiple rigid body transformations represented by unit dual quaternions $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_n$ with convex weights $\mathbf{w} = (w_1, \ldots, w_n)^T$:

$$DLB(\mathbf{w}; \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_n) = \frac{w_1\hat{\mathbf{q}}_1 + \ldots + w_n\hat{\mathbf{q}}_n}{\|w_1\hat{\mathbf{q}}_1 + \ldots + w_n\hat{\mathbf{q}}_n\|}$$

Obviously, the properties of DLB hold also in this more general case.

Our next goal is to achieve also constant speed solution and therefore fulfill all the properties from Section 4.2. For the case of 3D rotations, this can be done using spherical averages [22]. Unfortunately, it is not possible to simply apply spherical averages for unit dual quaternions, because the set of unit dual quaternions $\hat{Q}_1$ is not a hypersphere (instead, it is a set of tangent planes of a hypersphere [94]). Fortunately, it has been shown that Buss and Fillmore's idea [22] can be generalized to other manifolds [45]. The latter paper is based on the theory of matrix groups and Lie algebras. We propose a similar algorithm based on dual quaternions (see Algorithm 4.1), which is more efficient – due to the simple logarithm and exponential for dual quaternions (c.f. with the Rodriguez formula for rigid transformation matrices [101]).

> **Algorithm 4.1:** Dual quaternion iterative blending
> **Input:** Unit dual quaternions $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_n$, convex weights $\mathbf{w} = (w_1, \ldots, w_n)$, desired precision p
> **Output:** Blended unit dual quaternion $\hat{\mathbf{b}}$
> $DIB(\mathbf{w}; \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_n)$
> (1) $\hat{\mathbf{b}} = DLB(\mathbf{w}; \hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_n)$
> (2)   **repeat**
> (3) $\hat{\mathbf{x}} = \sum_{i=1}^{n} w_i \log(\hat{\mathbf{b}}^*\hat{\mathbf{q}}_i)$
> (4) $\hat{\mathbf{b}} = \hat{\mathbf{b}}\exp(\hat{\mathbf{x}})$
> (5)   **until** $\|\hat{\mathbf{x}}\| < p$
> (6)   **return** $\hat{\mathbf{b}}$

We leave a detailed mathematical discussion of this algorithm for a future work (considering the complexity of discussion of a simpler algorithm [22]). In practice, we observed the DIB algorithm to converge very quickly, typically in 1 to 4 steps. An intuitive explanation of this algorithm is shown in Figure 4.3.
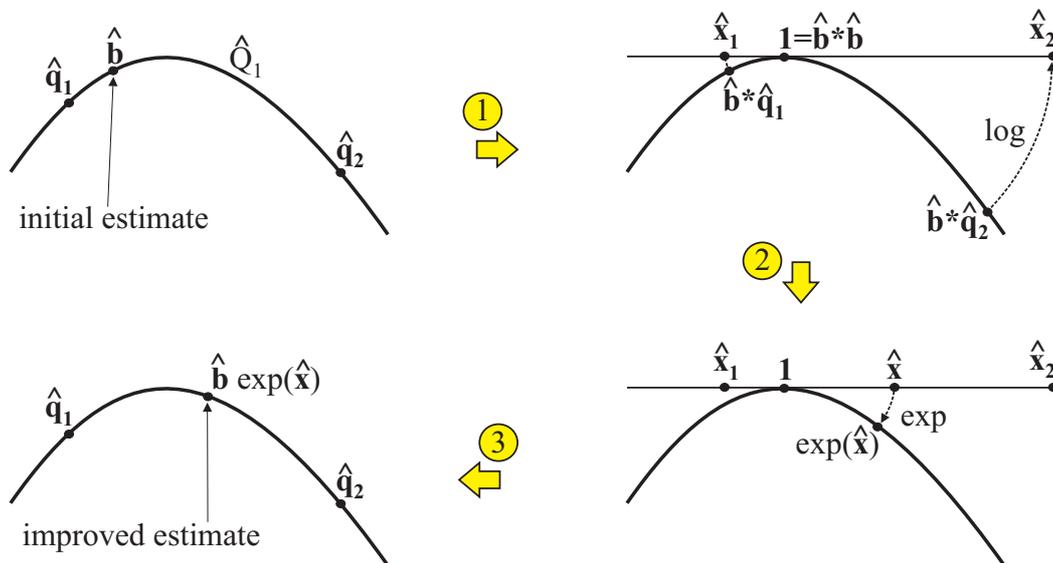
Figure 4.3: Illustration of one iteration of the DIB algorithm for $n = 2$ in a 2D slice of a 6-dimensional manifold $\hat{Q}_1$. First, the input dual quaternions are left-multiplied by $\hat{\mathbf{b}}^*$, which maps the initial estimate $\hat{\mathbf{b}}$ onto the identity. The logarithm mapping then transforms $\hat{\mathbf{b}}^*\hat{\mathbf{q}}_1$, $\hat{\mathbf{b}}^*\hat{\mathbf{q}}_2$ into the tangent space of $\hat{Q}_1$ at the identity, giving $\hat{\mathbf{x}}_1 = \log(\hat{\mathbf{b}}^*\hat{\mathbf{q}}_1)$, $\hat{\mathbf{x}}_2 = \log(\hat{\mathbf{b}}^*\hat{\mathbf{q}}_2)$. The blended value $\hat{\mathbf{x}} = w_1\hat{\mathbf{x}}_1 + w_2\hat{\mathbf{x}}_2$ is computed and projected back by the exponential mapping. Finally, multiplication $\hat{\mathbf{b}}\exp(\hat{\mathbf{x}})$ yields the unit dual quaternion closer to the exact solution.

The bi-invariance of DIB (and thus also the coordinate-invariance), follows from the bi-invariance of DLB and Lemma 4.3. The comparison of ScLERP with DIB is interesting. Not only is the result of DIB exactly equivalent to the result of ScLERP, but DIB finds this solution in just a single iteration (the second pass through the loop finds that $\|\hat{\mathbf{x}}\|$ is zero and performs no update of $\hat{\mathbf{b}}$). We prove this property in Lemma 4.7. First, however, we need the following:

Lemma 4.6. *For any unit dual quaternion* $\hat{\mathbf{q}}$ *and dual numbers* $\hat{a}, \hat{b}$, *it is true that* $\log(\hat{\mathbf{q}}^{\hat{a}}) = \hat{a}\log(\hat{\mathbf{q}})$ *and* $\hat{\mathbf{q}}^{\hat{a}}\hat{\mathbf{q}}^{\hat{b}} = \hat{\mathbf{q}}^{\hat{a}+\hat{b}}$.

Proof. The first statement is simple, because, by definition, $\log(\hat{\mathbf{q}}^{\hat{a}}) = \log(\exp(\hat{a}\log(\hat{\mathbf{q}}))) = \hat{a}\log(\hat{\mathbf{q}})$.

The second part is a little bit more tricky: by definition, $\hat{\mathbf{q}}^{\hat{a}}\hat{\mathbf{q}}^{\hat{b}} = \exp(\hat{a}\log\hat{\mathbf{q}})\exp(\hat{b}\log\hat{\mathbf{q}})$. However, we must be careful, because for dual (as well as regular) quaternions in general

$\exp(\hat{\mathbf{p}})\exp(\hat{\mathbf{q}}) \neq \exp(\hat{\mathbf{p}} + \hat{\mathbf{q}})$. Luckily, in our case, $\exp(\hat{a}\log\hat{\mathbf{q}})\exp(\hat{b}\log\hat{\mathbf{q}}) = \exp((\hat{a} + \hat{b})\log\hat{\mathbf{q}})$, as we show below. This is sufficient to finish the proof, because, by definition, $\exp((\hat{a} + \hat{b})\log\hat{\mathbf{q}}) = \hat{\mathbf{q}}^{\hat{a}+\hat{b}}$. According to Section 2.1.2, we can write $\log\hat{\mathbf{q}} = \hat{\mathbf{s}}\hat{c}$, where $\hat{\mathbf{s}}$ is a unit dual quaternion with zero scalar part. Note that $\hat{\mathbf{s}}\hat{\mathbf{s}} = (\mathbf{s}_0 + \epsilon\mathbf{s}_\epsilon)(\mathbf{s}_0 + \epsilon\mathbf{s}_\epsilon) = \mathbf{s}_0\mathbf{s}_0 + \epsilon(\mathbf{s}_\epsilon\mathbf{s}_0 + \mathbf{s}_0\mathbf{s}_\epsilon) = -1$, because of Formula (2.1) and the fact that $\langle\mathbf{s}_0, \mathbf{s}_\epsilon\rangle = 0$ (following from the unit length of $\hat{\mathbf{s}}$). This enables us to derive

$$\exp(\hat{\mathbf{s}}\hat{c}\hat{a})\exp(\hat{\mathbf{s}}\hat{c}\hat{b}) = (\cos\hat{c}\hat{a} + \hat{\mathbf{s}}\sin\hat{c}\hat{a})(\cos\hat{c}\hat{b} + \hat{\mathbf{s}}\sin\hat{c}\hat{b}) = \cos\hat{c}\hat{a}\cos\hat{c}\hat{b} - \sin\hat{c}\hat{a}\sin\hat{c}\hat{b} +$$
$$\hat{\mathbf{s}}(\sin\hat{c}\hat{a}\cos\hat{c}\hat{b} + \cos\hat{c}\hat{a}\sin\hat{c}\hat{b}) = \cos(\hat{c}\hat{a} + \hat{c}\hat{b}) + \hat{\mathbf{s}}\sin(\hat{c}\hat{a} + \hat{c}\hat{b}) = \exp(\hat{\mathbf{s}}\hat{c}(\hat{a} + \hat{b}))$$

<div align="right">□</div>

Lemma 4.7. *For the case of two rotations, the $DIB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$ algorithm converges in a single iteration with result $ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$.*

Proof. Thanks to the bi-invariance of DIB, we can perform the same simplification as when comparing DLB with ScLERP, that is instead of $DIB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$ and $ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$, compare $DIB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$ and $ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$. Let us denote the initial value of $\hat{\mathbf{b}}$ as $\hat{\mathbf{b}}_0 = DLB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$. With weight vector $\mathbf{w} = (1 - t, t)$ as the input (recalling convention from Section 4.2), the DIB computes in the first iteration $\hat{\mathbf{b}}_1 = \hat{\mathbf{b}}_0\exp((1 - t)\log(\hat{\mathbf{b}}_0^*) + t\log(\hat{\mathbf{b}}_0^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}))$. As shown above, the only difference between DLB and $ScLERP$ is in the angle of rotation and amount of translation. It means that $\hat{\mathbf{b}}_0 = (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{\hat{u}}$ for some dual number $\hat{u}$. Since for unit dual quaternions conjugation is the same as inverse, we can write, using Lemma 4.6,

$$\begin{aligned}
\hat{\mathbf{b}}_1 &= (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{\hat{u}}\exp((1 - t)\log((\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{-\hat{u}}) + t\log((\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{1-\hat{u}})) = (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{\hat{u}}\exp((t - \hat{u})\log(\hat{\mathbf{p}}^*\hat{\mathbf{q}})) \\
&= (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{\hat{u}+t-\hat{u}} = (\hat{\mathbf{p}}^*\hat{\mathbf{q}})^t = ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})
\end{aligned}$$

In the following iteration, the DIB algorithm computes

$$\begin{aligned}
\hat{\mathbf{x}} &= (1 - t)\log\hat{\mathbf{b}}_1^* + t\log(\hat{\mathbf{b}}_1^*\hat{\mathbf{p}}^*\hat{\mathbf{q}}) = (1 - t)\log((\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{-t}) + t\log((\hat{\mathbf{p}}^*\hat{\mathbf{q}})^{1-t}) \\
&= (t^2 - t + t - t^2)\log(\hat{\mathbf{p}}^*\hat{\mathbf{q}}) = 0
\end{aligned}$$

and thus the algorithm $DIB(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$ terminates with zero error and returns $\hat{\mathbf{b}}_1\exp(0) = \hat{\mathbf{b}}_1 = ScLERP(t; 1, \hat{\mathbf{p}}^*\hat{\mathbf{q}})$. Multiplication from left by $\hat{\mathbf{p}}$ and using the left-invariance yields $DIB(t; \hat{\mathbf{p}}, \hat{\mathbf{q}}) = ScLERP(t; \hat{\mathbf{p}}, \hat{\mathbf{q}})$. □

An immediate consequence is that DIB is also constant speed and shortest path. The DIB algorithm therefore meets all the properties from Section 4.2. However, it can be slower to compute than the closed-form DLB – the choice obviously depends on the needs of each particular application.

4.4 Comparison with Other Methods

This section summarizes the properties of previous blending algorithms and the dual quaternion based ones. Where applicable, we also state the execution time in FLOPS, see Table 4.1. Comments to the algorithms listed in Table 4.1 follow.

Linear – linear blending of homogeneous matrices is the most straightforward solution. As shown in Section 4.1, it is bi-invariant and naturally supports more than 2 transformations. Unfortunately, linear blending of rigid transformations does not preserve their rigidity, therefore also the constant speed and shortest path properties are not defined.

QLB_0' – based on the decomposition of a homogeneous matrix to a quaternion and a translation. Rigidity is preserved, because quaternion to matrix conversion always produces a valid rotation matrix. However, QLB_0' is not right invariant, as shown in Section 3.1, and therefore neither coordinate nor bi-invariant. Shortest path property of QLB_0' follows from Lemma 3.1 in Section 3.2 and its non-constant speed is discussed in the same section.

SB – spherical blending, i.e., $QLB_{\mathbf{r}_c}'$, where $\mathbf{r}_c$ is computed by least squares optimization (see Section 3.1). Therefore, spherical blending inherits the properties of QLB_0' and augments them with bi-invariance, as proven in Lemma 4.1 and Lemma 4.2.

Log – blending of matrix logarithms, proposed by Alexa [3], naturally preserves rigidity of input transformations, because of the properties of matrix exponential [101]. Invariance properties of log-matrix blending have been discussed in Section 4.1. As pointed out by Bloom et al. [13], log-matrix blending is not shortest path (even for pure rotations). Moreover, it is not constant speed, which we prove in the Appendix A.2.

Finally, the properties of our dual quaternion based blending algorithms, i.e., ScLERP, DLB and DIB, have been discussed in Section 4.3.

	Linear	QLB_0'	SB	Log	ScLERP	DLB	DIB
Preserving rigidity	-	+	+	+	+	+	+
Bi-invariance	+	-	+	-	+	+	+
Coord.-invariance	+	-	+	+	+	+	+
$n > 2$	+	+	+	+	-	+	+
Constant speed	-	-	-	-	+	-	+
Shortest path	-	+	+	-	+	+	+
FLOPS	$23n$	$28n + 26$	$(\star^1)$	$104n + 160$	240	$49n + 65$	$(\star^2)$

Table 4.1: Properties of different rigid transformation blending algorithms. $(\star^1)$ FLOPS for SB are determined by the applied SVD algorithm (which is orders of magnitude slower than the blending itself). $(\star^2)$ FLOPS for DIB depend on the actual input and on the desired precision.

We conclude that, for two rigid transformations, the optimal choice is ScLERP. If we need a precise solution for more than two transformations, we have to employ the iterative DIB. The FLOPS of log-matrix blending reported in Table 4.1 refer to an optimization, which is

restricted to rigid transformations and employs a closed-form Rodriguez formula for rigid transformations [101, 6]. However, even though it is closed-form, the Rodriguez formula is still more difficult to compute than the simple dual quaternion exp and log. We see that there is no reason to apply log-matrix blending for rigid transformations, because DLB has better properties and faster run-time. In the calculation of FLOPS, we assume that both input and output are 4×4 homogeneous matrices (specifically, the FLOPS for matrix$\leftrightarrow$dual quaternion conversions are included in Table 4.1; if an application works internally with quaternions instead of matrices, then the performance of our proposed algorithms is even better).

4.5   Application in Skinning

We have tested our DLB algorithm to the problem of skinning and have developed both CPU and GPU implementations. Having an existing code for linear blend skinning, this is quite straightforward (see Algorithm 4.2). In the pseudocode, function ROT-PART retrieves the rotational 3×3 submatrix of a homogeneous matrix and functions MATRIX2DQ, DQ2MATRIX convert between dual quaternions and homogeneous matrices (see Section 2.1.2).

> **Algorithm 4.2:** Dual quaternion skinning
> **Input:** $C_1, \ldots, C_p$ – joint transformation matrices for the current posture
> $\mathbf{v}_1, \ldots, \mathbf{v}_m$ – rest-pose vertices
> $\nu_1, \ldots, \nu_m$ – rest-pose normals
> $w_{1,1}, \ldots, w_{1,n_1}, \ldots, w_{m,1}, \ldots, w_{m,n_m}$ – vertex weights
> $j_{1,1}, \ldots, j_{1,n_1}, \ldots, j_{m,1}, \ldots, j_{m,n_m}$ – influencing joints indices
>   (see Section 2.2.1 for details)
> **Output:** $\mathbf{v}'_1, \ldots, \mathbf{v}'_m$ – vertices of the deformed skin
> $\nu'_1, \ldots, \nu'_m$ – normals of the deformed skin
> $\mathrm{DQS}(C_1, \ldots, C_p)$
> (1)   **for** $i = 1$ **to** p
> (2) $\hat{\mathbf{q}}_i = \mathrm{MATRIX2DQ}(C_i)$
> (3)   **for** $k = 1$ **to** m
> (4) $\hat{\mathbf{q}}_{blend} = DLB(w_{k,1}, \ldots, w_{k,n_k}; \hat{\mathbf{q}}_{j_{k,1}}, \ldots, \hat{\mathbf{q}}_{j_{k,n_k}})$
> (5) $M = \mathrm{DQ2MATRIX}(\hat{\mathbf{q}}_{blend})$
> (6) $\mathbf{v}'_k = M\mathbf{v}_k$
> (7) $\nu'_k = \mathrm{ROTPART}(M)\, \nu_k$

For an experimental evaluation, we used the woman model from Chapter 3 (just equipped with a more decent dress), having 5002 vertices, 9253 triangles and 54 joints. The cloth model used to illustrate the artifacts of SBS has 6000 vertices, 12000 triangles and 49 joints. The average run-time performance is reported in Figure 4.4.
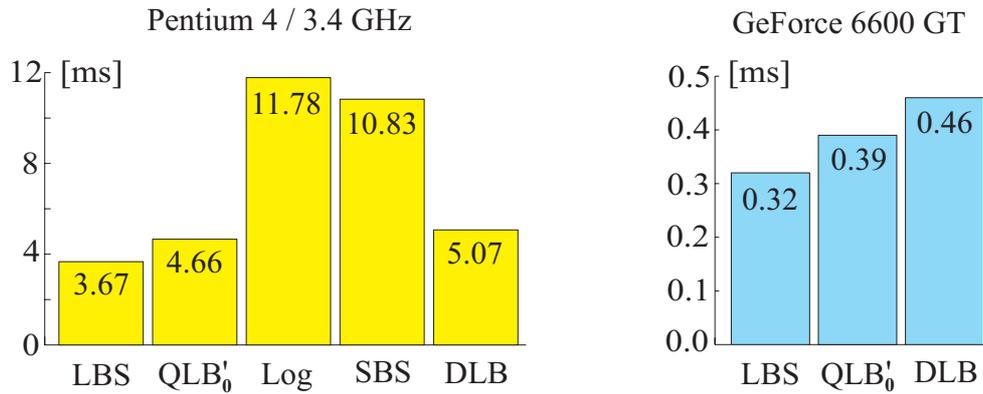
Figure 4.4: Average CPU/GPU run-times for skin deformation of a woman model in milliseconds.

We see that practical results are better than the theoretical ones presented in Table 4.1; this is because FLOPS counts in Table 4.1 include matrix to dual quaternion conversion, but our skinning application works internally with dual quaternions. The measurements, together with the visual results in Figure 4.5, clearly show the superiority of dual quaternion blending over previous methods: is not only more accurate, but also more than twice as fast as both log-matrix and spherical blending.

Dual quaternion skinning is slightly slower than linear blend skinning and QLB'_0, but we believe that this is not a high price to pay for the elimination of artifacts. When compared to linear blending, dual quaternions have one more advantage: they only need 8 floats instead of the 12 required by matrices, which is of particular benefit for the GPU implementation.

Figure 4.5: Comparison of dual quaternion skinning with (a) linear blend skinning, (b) log-matrix blending, (c) direct quaternion blending – QLB'_0, (d) spherical blend skinning.

5 Collision Detection for Linear Blend Skinning

In this chapter, we discuss the problem of collision detection (CD) with an object deformed by linear blend skinning (see Section 2.2.1). The reason to consider linear blend skinning (while we presented more advantageous blending methods in the previous chapters) is that CD is not trivial already in the case of linear blending. Therefore, we find it natural to start with this simplest skinning algorithm. Also, the collision detection for spherical blend skinning, presented in Chapter 6, builds on the results from this chapter.

It would be of course possible to apply a general CD algorithm (see Section 2.3.2) for models deformed by linear blend skinning. However, this is not the most efficient way, because no general deformable CD algorithm can work in a sub-linear time with respect to the number of vertices. This is obvious: if general deformations are supported, it is necessary to visit each vertex – just to find out the deformation.

This is a serious drawback when compared to the rigid body CD, which can be processed in a sub-linear time thanks to bounding volume hierarchies. Sub-linear CD algorithms for special deformation models have been presented in [81, 62]. They are based on an on-demand refitting of bounding volumes, which enables us to achieve sub-linear time complexity. Unfortunately, both [81, 62] work only with linear combinations of vertex displacements (see Section 2.3.2) and therefore cannot be applied to linear blend skinning.

In this chapter, we discuss how to design an on-demand refitting for linear blend skinning and thus obtain a sub-linear CD algorithm.

5.1   Sphere Tree Construction

We have chosen the most simple bounding volume – a sphere, because of its one unique feature: invariance under rotation. This property is very advantageous for the on-demand refitting operation. Sphere tree hierarchy is constructed for the skin in the reference position.

Since our goal is a precise algorithm, we must ensure that each triangle of the skin is covered by a bounding sphere entirely. (We do not allow covering of one triangle by several bounding spheres, although this could be advantageous for long thin triangles.) In the first pass, we construct only a binary sphere tree according to Algorithm 2.1. We compute the minimal enclosing sphere for a set of vertices in each node using a fast exact algorithm [36]. This ensures that our spheres in the reference position are as tight as possible.

After the initial sphere tree is built, we execute a second pass which optimizes the sphere tree. We observed that it may happen that the radii of the children and parent spheres do not differ considerably (see example in Section 5.4). If a node c has a sphere of similar size as its parent node p, it means that it has only a small discriminating power for the CD query. We can therefore remove the node c from the tree and assign its children directly to node p. It does not violate the correctness of the algorithm, but saves an intersection

test and several refitting operations. Therefore, in the second pass, we collapse the binary tree to a general n-ary tree, following a simple rule. Let us denote the radii of spheres in nodes c and p as c_r and p_r. The node c is deleted if

$$c_r > C \cdot p_r$$

where C is a user-defined constant. We found the number $C = 0.6$ to work well in practice.

5.2   On-demand Sphere Refitting

Recall that linear blend skinning (LBS) is based on Formula (2.6). In the following, we call the joints $j_1, \ldots, j_n$ influencing vertex $\mathbf{v}$ as its *joint-set* and denote it as $J(\mathbf{v})$.

In this section, we describe an efficient method for on-demand sphere refitting, which is the key part of our sub-linear CD algorithm. Our goal is to detect collisions between models deformed by LBS in an arbitrary posture. During pre-processing, a sphere tree for the model in the reference position is built according to Section 5.1. During the CD query, it is necessary to transform the bounding spheres from the reference position to the current (animated) posture preserving their bounding property, i.e., ensuring that a transformed sphere encloses all the geometry that it enclosed in reference position. It does not affect the result of CD if the transformed sphere is bigger than necessary, although it might affect the performance.

Observe that the only thing that changes the shape of the deformed skin during the animation are the joint rotations, i.e., the transformation matrices C_{j_i}. The number of joints in a typical model is much smaller than the number of vertices. The main idea of our on-demand refitting operation is to update the bounding spheres using only the joint transformations C_{j_i} (and some pre-computed information which is invariant during the animation). As mentioned in Section 2.2.1, the LBS does nothing else but a convex combination of individual vertex transformations. In order to enclose the transformed geometry, it is sufficient to enclose only the transformed vertices (since triangles are convex).

Assume that we are refitting a sphere S containing vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$ in the reference posture. The list of all joint-sets influencing $\mathbf{v}_1, \ldots, \mathbf{v}_t$ is pre-computed and stored in the node representing sphere S. For simplicity, we first assume that all vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$ are influenced by only one joint-set J, i.e., $J = J(\mathbf{v}_1) = J(\mathbf{v}_2) = \ldots = J(\mathbf{v}_t)$.

LBS transforms vertex $\mathbf{v}_i$ to $\mathbf{v}_i'$ according to Formula (2.6). From this equation it can be seen immediately that

$$\mathbf{v}_i' \in CH(\{C_j \mathbf{v}_i : j \in J\})$$

Since $\mathbf{v}_i \in S$, it indicates that we can transform the whole sphere S instead of individual vertices. To transform a sphere by a homogeneous matrix it is sufficient to transform only the center of the sphere and keep the radius intact (thanks to the rotation invariance).

Since $C_j\mathbf{v}_i \in C_jS$ for any $j \in J$, we can write

$$\mathbf{v}'_i \in CH\left(\bigcup_{j \in J} C_jS\right) \tag{5.1}$$

for any $i = 1, \ldots, t$. Note that the bounding volume in Formula (5.1) depends only on the bounding sphere S and the current joint transformations C_j, i.e., it can be computed in time sub-linear to l – the number of vertices. In practice, we work with minimal enclosing sphere instead of convex hull. This is correct indeed, because sphere is a convex set and therefore contains the convex hull.

Unfortunately, the bounding volume according to Formula (5.1) is very loose (conservative). This is especially apparent if sphere S contains only few vertices. The bad approximation quality of $CH\left(\bigcup_{j \in J} C_jS\right)$ follows from the fact that it contains points $\mathbf{v}'_i$ for *arbitrary* convex weights. In other words, it encloses not only the current LBS deformation, but all possible LBS deformations that could be achieved by varying vertex weights. We can make the bounding spheres much more tight by taking the actual model's vertex weights into account.

This is illustrated in Figure 5.1. In the left image, we see three vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$ bounded by sphere S in the reference position. These vertices are influenced by joints j_1 and j_2, let us say that the weights of $\mathbf{v}_1$ are $0.6, 0.4$ (for j_1 and j_2), of $\mathbf{v}_2$: $0.5, 0.5$, and of $\mathbf{v}_3$: $0.4, 0.6$ (indicated by color). The middle image shows the animated skeleton. All possible positions of the deformed vertex $\mathbf{v}'_i$ form set

$$L_i = \{wC_{j_1}\mathbf{v}_i + (1 - w)C_{j_2}\mathbf{v}_i : w \in [0, 1]\}, \quad i = 1, 2, 3$$

which are the line segments illustrated in the picture. The bounding volume BV_1 given as the convex hull of transformed spheres $C_{j_1}S$ and $C_{j_2}S$ encloses the lines L_1, L_2, L_3 and therefore also the new vertex positions $\mathbf{v}'_1, \mathbf{v}'_2, \mathbf{v}'_3$. A smaller bounding volume BV_2 is depicted in Figure 5.1 right. It is created by considering that vertex weights for j_1 fall into interval $[0.4, 0.6]$. Therefore, it is sufficient to enclose only shorter line segments

$$\{wC_{j_1}\mathbf{v}_i + (1 - w)C_{j_2}\mathbf{v}_i : w \in [0.4, 0.6]\}$$

which gives BV_2. The next section describes how to exploit this observation to create tighter refitted spheres.

5.2.1 Optimized Sphere Refitting

For notational simplicity, let us assume that all vertices enclosed by S belong to the joint-set $J = \{1, \ldots, n\}$. For any $j \in J$ we define the smallest interval of weights $[l_j, h_j]$ such that all vertex weights of joint j fit into this interval. These weight intervals are used to create smaller refitted spheres. The following construction is based on a concept of convex
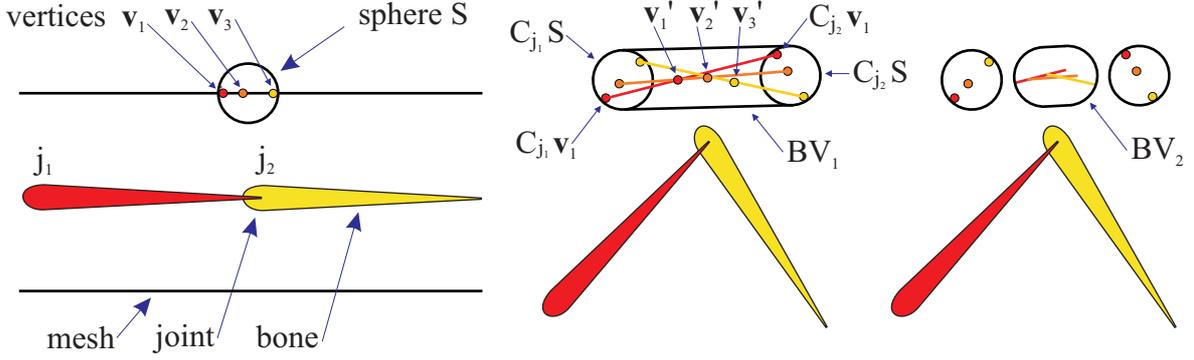
Figure 5.1: Left: the reference posture; middle and right: the animated posture. C_{j_1} (C_{j_2}) is the transformation matrix of joint j_1 (j_2). Bounding volume BV_1 is bigger than necessary, because it encloses all possible deformations of vertices $\mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3$. Our algorithm uses an optimized bounding volume BV_2, which considers actual vertex weights.

combination of spheres – a generalization of convex combination of points. For brevity, we denote the transformed spheres as $S_j = C_j S$ for each $j = 1, \dots, n$. We define the convex combination of spheres $S_1, \dots S_n$ (or any general convex sets) with weight vector $\mathbf{w} \in W_n$ as:

$$\sum_{i=1}^{n} w_i S_i \equiv \left\{ \sum_{i=1}^{n} w_i \mathbf{x}_i : \mathbf{x}_i \in S_i \right\}$$

i.e., the set of convex combinations of all points from $S_1, \dots, S_n$. This definition is a natural extension of standard convex combination of points. The following lemma claims that a convex combination of spheres is a sphere which can be computed as a convex combination of sphere centers and radii.

Lemma 5.1. *Let $S_1, \dots S_n$ be spheres in R^d with centers $\mathbf{s}_i \in R^d$ and radii r_i. If $\mathbf{w} \in W_n$ then*

$$\sum_{i=1}^{n} w_i S_i = S$$

where S is a sphere with center $\mathbf{s} = \sum_{i=1}^{n} w_i \mathbf{s}_i$ and radius $r = \sum_{i=1}^{n} w_i r_i$.

Proof. Let $\mathbf{x} \in \sum_{i=1}^{n} w_i S_i$. According to the definition of convex combination of convex sets, we have $\mathbf{x}_i \in S_i$ such that $\mathbf{x} = \sum_{i=1}^{n} w_i \mathbf{x}_i$. Obviously $\|\mathbf{x}_i - \mathbf{s}_i\| \leq r_i$ for each $i = 1, \dots n$. Multiplying the equations by w_i and summing them together yields

$$\sum_{i=1}^{n} \|w_i \mathbf{x}_i - w_i \mathbf{s}_i\| \leq \sum_{i=1}^{n} w_i r_i$$

The triangle inequality says that

$$\left\| \sum_{i=1}^{n} w_i \mathbf{x}_i - \sum_{i=1}^{n} w_i \mathbf{s}_i \right\| \leq \sum_{i=1}^{n} \|w_i \mathbf{x}_i - w_i \mathbf{s}_i\|$$

Putting both inequalities together gives

$$\|\mathbf{x} - \mathbf{s}\| = \left\| \sum_{i=1}^{n} w_i \mathbf{x}_i - \sum_{i=1}^{n} w_i \mathbf{s}_i \right\| \leq \sum_{i=1}^{n} w_i r_i = r$$

which shows that $\mathbf{x} \in S$, and therefore $\sum_{i=1}^{n} w_i S_i \subseteq S$. In order to show the opposite inclusion, choose $\mathbf{x} \in S$, i.e., satisfying $\|\mathbf{x} - \mathbf{s}\| \leq r$. Consider points

$$\mathbf{x}_i = \frac{(\mathbf{x} - \mathbf{s}) r_i}{r} + \mathbf{s}_i$$

for $i = 1, \ldots n$. Note that $\mathbf{x}_i \in S_i$, because

$$\|\mathbf{x}_i - \mathbf{s}_i\| = \left\| \frac{(\mathbf{x} - \mathbf{s}) r_i}{r} \right\| \leq \frac{r \, r_i}{r} = r_i$$

Moreover,

$$\sum_{i=1}^{n} w_i \mathbf{x}_i = \frac{\mathbf{x} - \mathbf{s}}{r} \sum_{i=1}^{n} w_i r_i + \sum_{i=1}^{n} w_i \mathbf{s}_i = \frac{(\mathbf{x} - \mathbf{s}) r}{r} + \mathbf{s} = \mathbf{x}$$

which shows that $\mathbf{x} \in \sum_{i=1}^{n} w_i S_i$, and therefore also $S \subseteq \sum_{i=1}^{n} w_i S_i$. □

Lemma 5.1 is interesting by itself, but we use it also to show that it is possible to rewrite the bounding volume from Formula (5.1) to

$$CH \left(\bigcup_{j \in J} S_j \right) = \bigcup_{\mathbf{w} \in W_n} \sum_{j=1}^{n} w_j S_j \tag{5.2}$$

which is a generalization of the well-known identity for convex hulls of points. In order to show the correctness of Formula (5.2), we prove first the convexity of the set on the right hand side of Formula (5.2).

Lemma 5.2. *Let $S_1, \ldots S_n$ are spheres in R^d. Then the set*

$$M = \bigcup_{\mathbf{w} \in W_n} \sum_{j=1}^{n} w_j S_j$$

is convex.

Proof. Let us assume that sphere S_i has center $\mathbf{s}_i \in R^d$ and radius r_i. We fix an arbitrary $\mathbf{x} \in M$, $\mathbf{x}' \in M$ and $\lambda \in \langle 0, 1 \rangle$, and we have to proof that $\mathbf{x}'' = (1 - \lambda)\mathbf{x} + \lambda \mathbf{x}' \in M$. By the definition of M, there exist weight vectors $\mathbf{w} \in W_n$ and $\mathbf{w}' \in W_n$ such that $\mathbf{x} \in \sum_{i=1}^{n} w_i S_i$ and $\mathbf{x}' \in \sum_{i=1}^{n} w_i' S_i$. According to Lemma 5.1, it means that

$$\left\| \mathbf{x} - \sum_{i=1}^{n} w_i \mathbf{s}_i \right\| \leq \sum_{i=1}^{n} w_i r_i, \quad \left\| \mathbf{x}' - \sum_{i=1}^{n} w_i' \mathbf{s}_i \right\| \leq \sum_{i=1}^{n} w_i' r_i$$

Consider weights

$$w_i'' = (1 - \lambda)w_i + \lambda w_i', \quad i = 1, \ldots, n$$

and observe that $\mathbf{w}'' \in W_n$: obviously $w_i'' \geq 0$ and $\sum_{i=1}^n w_i'' = (1-\lambda)\sum_{i=1}^n w_i + \lambda \sum_{i=1}^n w_i' = 1$. Using the triangle inequality we derive

$$\left\| \mathbf{x}'' - \sum w_i'' \mathbf{s}_i \right\| = \left\| (1-\lambda)\mathbf{x} - \sum(1-\lambda)w_i\mathbf{s}_i + \lambda\mathbf{x}' - \sum \lambda w_i'\mathbf{s}_i \right\| \leq$$

$$(1-\lambda)\left\| \mathbf{x} - \sum w_i\mathbf{s}_i \right\| + \lambda \left\| \mathbf{x}' - \sum w_i'\mathbf{s}_i \right\| \leq (1-\lambda)\sum w_i r_i + \lambda \sum w_i' r_i = \sum w_i'' r_i$$

Again by Lemma 5.1, this means that $\mathbf{x}'' \in \sum_{i=1}^n w_i'' S_i$ and thus also $\mathbf{x}'' \in M$. $\qquad \square$

In order to verify Formula (5.2), it is now sufficient to show the following:

Lemma 5.3. *Let $S_1, \ldots S_n$ are spheres in R^d. Then the set $M = \bigcup_{\mathbf{w} \in W_n} \sum_{j=1}^n w_j S_j$ is the convex hull of $S_1 \cup \cdots \cup S_n$.*

Proof. By Lemma 5.2 we know that the set M is convex, and M obviously contains $S_1, \ldots, S_n$. Therefore, it is sufficient to show that if any other convex set $C \subseteq R^d$ contains $S_1, \ldots S_n$, then it contains also M. To show that C contains M, it is sufficient to show that $\sum_{i=1}^n w_i S_i \subseteq C$ for any $\mathbf{w} \in W_n$. Let us choose $\mathbf{x} \in \sum_{i=1}^n w_i S_i$. By definition, for any $i = 1, \ldots, n$ we have $\mathbf{x}_i \in S_i$ such that $\mathbf{x} = \sum_{i=1}^n w_i \mathbf{x}_i$. We assumed $S_i \subseteq C$ which means that $\mathbf{x}_i \in C$ for each $i = 1, \ldots, n$. The convexity of C assures $\mathbf{x} \in C$. This verifies that $\sum_{i=1}^n w_i S_i \subseteq C$. $\qquad \square$

Having verified Formula (5.2), we can take the weight intervals into account by defining $W_n' = \{\mathbf{x} \in W_n : l_i \leq x_i \leq h_i, \ i = 1, \ldots n\}$ and changing the bounding volume to

$$M' = \bigcup_{\mathbf{w} \in W_n'} \sum_{j=1}^n w_j S_j \tag{5.3}$$

This is correct, because only the weights that actually appear among the bounded vertices are important, and all these weights are within $[l_1, h_1] \times \cdots \times [l_n, h_n]$. Because of Formula (5.2), we refer to the set M' as to the *generalized convex hull* of spheres $S_1, \ldots, S_n$. It is obvious that the generalized convex hull is always a subset of the non-generalized one, and they are equal if and only if all weight intervals are $[0, 1]$.

The resulting refitted sphere computed by our algorithm is therefore the bounding sphere of M'. From the definition it is not straightforward to see how M' looks like, and how its enclosing sphere could be efficiently computed. This is an essential part of the proposed algorithm and therefore deserves our attention.

The main trick is that instead of working directly with spheres $S_1, \ldots, S_n$, we compute another set of spheres $R_1, \ldots, R_m$, whose ordinary (non-generalized) convex hull will be

equivalent to the generalized convex hull of spheres $S_1, \ldots, S_n$. The spheres $R_1, \ldots, R_m$ reduce the problem to computing the minimal enclosing sphere of spheres.

Our task now is to transform the set of spheres $S_1, \ldots, S_n$ with weight intervals $[l_1, h_1]$, $\ldots, [l_n, h_n]$ into spheres $R_1, \ldots, R_m$, turning the generalized convex hull into a standard one. Let us examine the set W'_n. Assuming naturally $0 \leq l_i \leq h_i \leq 1$, the set W'_n can be written explicitly as

$$W'_n = \{\mathbf{w} \in R^n : l_i \leq w_i \leq h_i, \ i = 1, \ldots n, \ \sum_{i=1}^{n} w_i = 1\}$$

We can interpret W'_n geometrically as an intersection of $2n$ half-spaces and one hyperplane. It means that W'_n is a bounded $(n-1)$-dimensional convex set in R^n, and therefore, it can be expressed as a convex hull of some points in R^n (because of the equivalency of bounded H-polytopes and V-polytopes [93]). We call these points *corners* and denote them as $\mathbf{c}_1, \ldots, \mathbf{c}_m$. Since the set W'_n depends only on constant vertex weights, the corners can be pre-computed and stored along with the model. We employed only a simple brute-force computation of corners: testing all intersections and discarding those outside $[l_1, h_1] \times \ldots \times [l_n, h_n]$. Once the corners are known, the spheres $R_1, \ldots, R_m$ can be computed as

$$R_i = \sum_{j=1}^{n} c_{i,j} S_j, \quad i = 1, \ldots, m \tag{5.4}$$

In order to justify this formula, consider that W'_n is the convex hull of $\mathbf{c}_1, \ldots, \mathbf{c}_m$, and thus can be expressed as $W'_n = \{\sum_{i=1}^{m} u_i \mathbf{c}_i : \mathbf{u} \in W_m\}$. Therefore

$$\bigcup_{\mathbf{w} \in W'_n} \sum_{j=1}^{n} w_j S_j = \bigcup_{\mathbf{u} \in W_m} \sum_{j=1}^{n} \left(\sum_{i=1}^{m} u_i c_{i,j} \right) S_j$$

because the j-th component of $\sum_{i=1}^{m} u_i \mathbf{c}_i$ is $\sum_{i=1}^{m} u_i c_{i,j}$. Since the convex combination of spheres is nothing but a convex combination of their centers and radii, we can swap the sums

$$\bigcup_{\mathbf{u} \in W_m} \sum_{j=1}^{n} \left(\sum_{i=1}^{m} u_i c_{i,j} \right) S_j = \bigcup_{\mathbf{u} \in W_m} \sum_{i=1}^{m} u_i \left(\sum_{j=1}^{n} c_{i,j} S_j \right)$$

and $\left(\sum_{j=1}^{n} c_{i,j} S_j \right)$ is the new sphere R_i. Putting the equations together we see that

$$\bigcup_{\mathbf{w} \in W'_n} \sum_{i=1}^{n} w_i S_i = \bigcup_{\mathbf{u} \in W_m} \sum_{i=1}^{m} u_i R_i$$

This shows that the generalized convex hull of spheres $S_1, \ldots, S_n$ is really equivalent to the ordinary convex hull of spheres $R_1, \ldots, R_m$.

An example is presented in Figure 5.2. In the picture, CH_1 is the convex hull of spheres S_1, S_2, S_3. CH_2 is the generalized convex hull of spheres S_1, S_2, S_3 with respect to weight intervals $[0, 0.6]$, $[0, 1]$, $[0, 1]$. The right part of the picture demonstrates that CH_2 can be also expressed as $CH(R_1, R_2, R_3, R_4)$ where spheres R_1, R_2, R_3, R_4 are computed according to Formula (5.4).
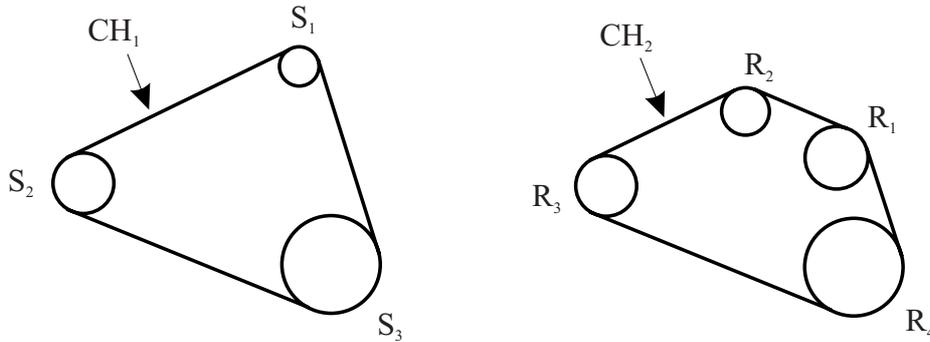
Figure 5.2: The standard and generalized convex hull of spheres S_1, S_2, S_3. The generalized convex hull of S_1, S_2, S_3 is equivalent to the ordinary convex hull of spheres R_1, R_2, R_3, R_4.

5.3   Algorithm Overview

In this section we summarize our CD algorithm for models deformed by LBS. First, we make some off-line pre-processing for every model. The pre-processing involves building the sphere tree, as discussed in Section 5.1. For each bounding sphere S in the tree we perform some further precomputations. Let us assume that sphere S encloses vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$. First of all, we determine the joint-sets $J_1, \ldots, J_k$ that cover all the joint-sets within $\mathbf{v}_1, \ldots, \mathbf{v}_t$, i.e., such that $J_1 \cup \ldots \cup J_k = J(\mathbf{v}_1) \cup \ldots \cup J(\mathbf{v}_t)$. We discard potential duplicates, ensuring that $J_i \neq J_j$ for each $i \neq j$.

Let p_i denote the number of joints in joint-set J_i. For each $J_i = \{j_{i,1}, \ldots, j_{i,p_i}\}$, we compute the weight intervals as follows. We start with empty weight intervals and check all vertices associated with J_i. Their weights are included into $[l_{i,1}, h_{i,1}], \ldots, [l_{i,p_i}, h_{i,p_i}]$, inflating the intervals if necessary. Then we compute the corners $\mathbf{c}_{i,1}, \ldots, \mathbf{c}_{i,m_i}$ of the resulting weight bound, as indicated in the previous section. We store the corners and the joint-sets in the final tree, while the vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$ need not be stored there.

When processing a collision detection query for animated (deformed) models, we apply Algorithm 2.2 from Section 2.3.1. We modify this algorithm only by adding the refitting operation, which is inserted just prior to the sphere intersection test. This ensures that we work with correct bounding spheres even though the model has changed its shape.

In order to refit bounding sphere S, we consider each of its joint-sets $J_1, \ldots, J_k$. For each $J_i = \{j_{i,1}, \ldots, j_{i,p_i}\}$ we transform S by the transformations of joints $j_{i,1}, \ldots, j_{i,p_i}$. It results

in spheres $S_{i,1} = C_{j_{i,1}}S, \ldots, S_{i,p_i} = C_{j_{i,p_i}}S$, which are blended using Formula (5.4) and corners $\mathbf{c}_{i,1}, \ldots, \mathbf{c}_{i,m_i}$. The result of blending is another set of spheres: $R_{i,1}, \ldots, R_{i,m_i}$. We merge together these spheres for all joint-sets ($i = 1, \ldots, k$). Finally, we enclose spheres $R_{1,1}, \ldots, R_{k,m_k}$ by a single bounding sphere, which is the result of the refitting operation. Note that according to the previous section, already $CH(R_{1,1}, \ldots, R_{k,m_k})$ bounds the current mesh deformation. We use only a simple approximation of minimal enclosing sphere of spheres, see Algorithm 5.1. The same approximation of minimal enclosing spheres has been used in [62], where it has been found that these conservative bounding spheres result in faster CD than exact but slower to compute minimal enclosing spheres. In the algorithm pseudocode, we denote a sphere with center $\mathbf{s}$ and radius r as $(\mathbf{s}, r)$.

Algorithm 5.1: Bounding sphere of spheres
Input: $L = (\mathbf{s}_1, r_1), \ldots, (\mathbf{s}_n, r_n)$ – list of spheres
Output: $(\mathbf{s}_e, r_e)$ – enclosing sphere of spheres in L
BOUNDINGSPHERE(L)
(1) $\mathbf{s}_e = (\sum_{k=1}^n \mathbf{s}_i)/n$
(2) $r_e = 0$
(3)   **for** $k = 1$ **to** n
(4)   **if** ($\|\mathbf{s}_k - \mathbf{s}_e\| + r_k > r_e$)
(5) $r_e = \|\mathbf{s}_k - \mathbf{s}_e\| + r_k$
(6)   **return** $(\mathbf{s}_e, r_e)$

For the final sphere refitting pseuodocode, see Algorithm 5.2.

Algorithm 5.2: Sphere refitting for linear blend skinning
Input: S – sphere to be refitted (expressed in the reference posture)
          $C_1, \ldots, C_p$ – joint transformation matrices for the current posture
          $J_1, \ldots, J_k$ – joint-sets influencing sphere S
          $\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,m_1}, \ldots, \mathbf{c}_{k,1}, \ldots, \mathbf{c}_{k,m_k}$ – pre-computed corners
Output: sphere S refitted for the current skin deformation
LBSSPHEREREFIT(S)
(1)   **for** $i = 1$ **to** k
(2)   **for** $h = 1$ **to** p_i
(3) $S_{i,h} = C_{j_{i,h}}S$
(4)   **for** $h = 1$ **to** m_i
(5) $R_{i,h} = \sum_{j=1}^{p_i} c_{i,h,j}S_{i,j}$
(6)   **return** BOUNDINGSPHERE($R_{1,1}, \ldots, R_{k,m_k}$)

5.4   Results and Comparison

We tested our collision detection algorithm on three models, whose complexities are described in Table 5.1. First, we examine the tightness of bounding spheres. Before the tree

	Vertices	Triangles	Joints
Dwarf	859	1664	45
Man	4435	8270	27
Creature	6682	13590	56

Table 5.1: Complexities of example models

optimization, the total number of spheres for the creature model is 27079 and the sphere tree depth is 22. The average radii of spheres for each level are: 34.55 31.82 23.63 18.86 13.07 7.32 4.86 3.46 2.60 1.92 1.47 1.14 0.90 0.72 0.61 0.53 0.47 0.42 0.39 0.38 0.37 0.36. Note that the average radii on certain neighboring levels are close to each other. This is improved by collapsing the tree according to Section 5.1. After this step (with $C = 0.6$) only 18679 spheres remain in the tree and its depth decreases to 9, while the number of children increases from 2 to 15 (in an extremal case). The resulting sphere tree is presented in Figure 5.3.

The average radii of the resulting spheres in the reference position can be found in the second column of Table 5.2 (Reference). The third column of Table 5.2 (On-demand) lists average radii of spheres refitted by our algorithm for the animated posture (Figure 5.3 bottom). Results for the bottom-up refitted spheres are in the fourth column. The average radii of minimal enclosing spheres are reported in the last column of Table 5.2. They are computed directly from the deformed vertices and thus represent the best possible result of any refitting. From Table 5.2, as well as from Figure 5.3, it as apparent that

Level	Reference	On-demand	Bottom-up	Best
1	34.55	43.89	45.68	33.33
2	18.86	26.17	24.63	19.39
3	7.38	8.05	9.21	7.31
4	3.68	3.95	4.58	3.70
5	1.69	1.77	2.27	1.70
6	0.91	0.94	1.17	0.92
7	0.61	0.63	0.74	0.61
8	0.42	0.43	0.49	0.42
9	0.30	0.30	0.35	0.30

Table 5.2: Average radii of spheres on each level of the sphere tree for the creature model. The column "Reference" considers spheres in the reference position, while the other columns refer to spheres in the animated posture.

our on-demand refitting operation produces quite tight bounding spheres. On most levels,
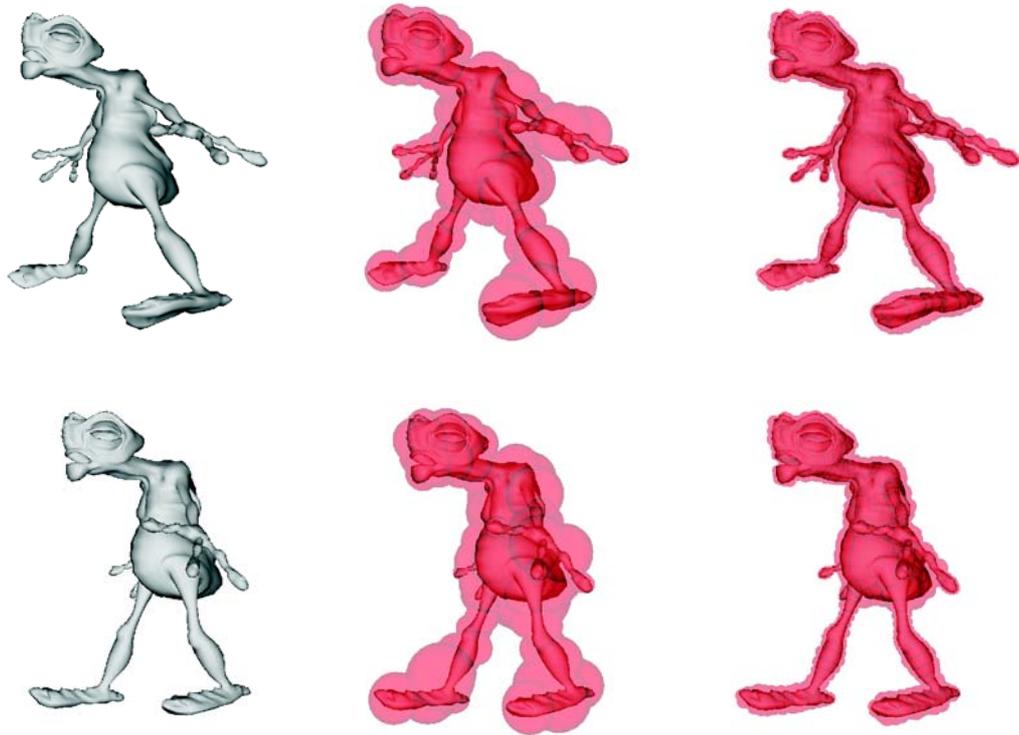
Figure 5.3: **Top:** the reference posture of the creature model with spheres on levels 4 and 6 of the tree (pre-computed according to Section 5.1), **Bottom:** an animated posture with spheres refitted by our algorithm (during run-time).

bottom-up refitting is outperformed by the on-demand refitting, which is actually not far from the optimal solution. Notice that the gap between the on-demand and best possible results is smaller in higher levels. This is understandable, because the on-demand refitted spheres deep in the tree are typically only transformed reference spheres, which are optimal (see Section 5.1).

We tested the collision detection run-time performance on a 2.5GHz Athlon CPU under normal working conditions. In the first scenario, two men are walking towards each other. One frame of the animation is presented in Figure 5.4, together with spheres refitted by our algorithm. Our algorithm detects all collisions in average 0.27ms per frame, while the algorithm using the complete bottom-up refitting needs 13.6ms. The about 50 times faster execution is achieved by refitting only the spheres that are important for the CD query. Our algorithm refits only the blue spheres in Figure 5.4, while the bottom-up refitting has to refit all spheres, both blue and red. The time for the bottom-up refitting itself is 12.9ms, which reveals that this is the bottleneck of the CD indeed. The refitting of all the 15339 spheres by our on-demand refitting operation takes 15.16ms (less than $1\mu s$ per sphere). This shows that our refitting operation is quite fast in practice.
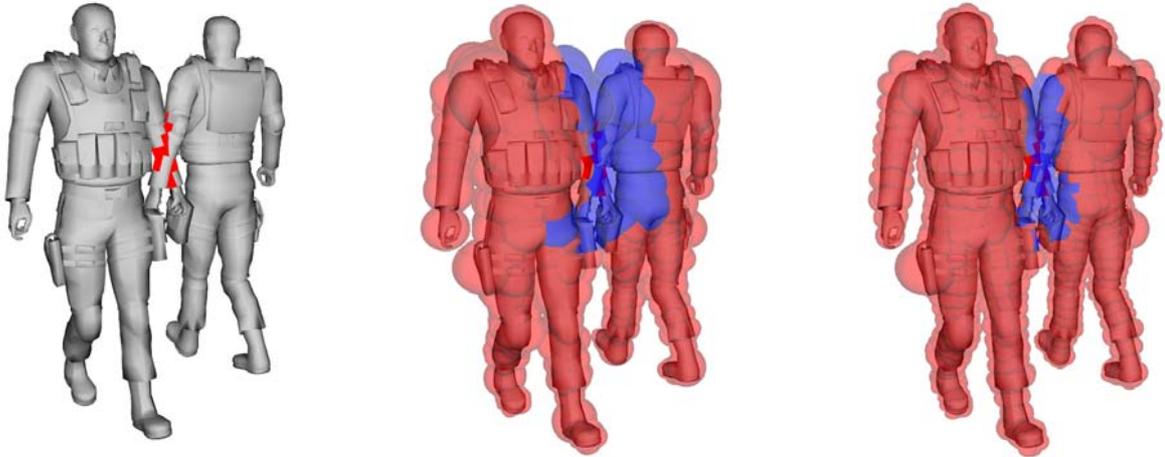
Figure 5.4: A collision of two man models with spheres on level 5 and 6 of the tree. Our lazy algorithm does not refit the red spheres, in contrast to eager bottom-up refitting.

The previous scenario resulted in great speed-up because the number of intersections was not big. In order to test the algorithm in a more complex situation, we designed a torture test: two creatures walking through each other, see Figure 5.5. Such a scenario is unlikely to occur in practice, because collision response algorithms usually prevent large interpenetrations. In this animation, the average time of our CD test is 6.14ms, and the CD using
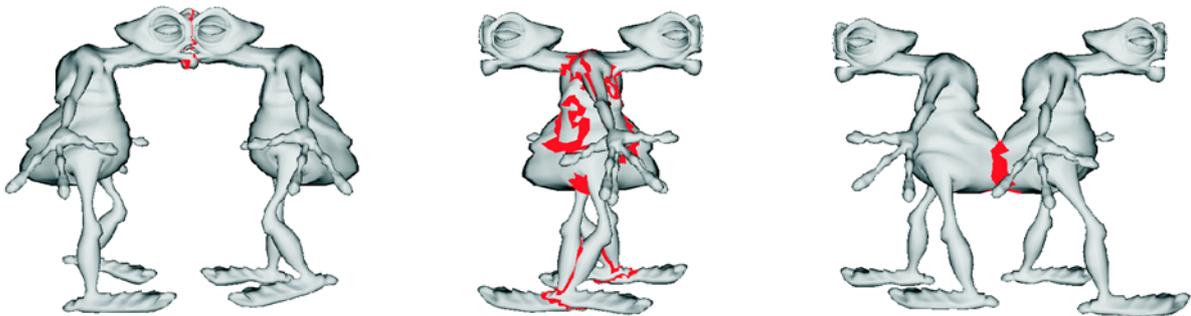
Figure 5.5: Torture test for our algorithm, involving lot of collisions.

bottom-up refitting takes 32.7ms. The speed-up of our approach is not as high as in the previous example, but shows that the on-demand refitting still performs much better than the bottom-up one.

Some applications do not need to determine the whole set of colliding triangles – the information whether the models are intersecting is sufficient, e.g., for backtracking to a collision-free state. In such a situation, the CD algorithm can stop after finding the first intersecting triangle pair. The CD algorithm with our on-demand refitting operation benefits greatly from this fact, unlike the complete bottom-up refitting. In this case, the average

time of our CD test for the same scenario drops down to 0.72ms, while the CD with the bottom-up refitting time improves only slightly: to 26.5ms.

Another important aspect of a CD algorithm is its scalability, i.e., how its performance changes when the size of input data increases. We executed a test on a pair of dwarf models in an animated posture, both in collision free and interfering position, see Figure 5.6. Starting with a low-polygonal model, we subdivided each triangle to four smaller ones, which ensures the same shape for all CD queries. The results for the colliding situation

Figure 5.6: The meshes of dwarf models were subdivided several times in order to evaluate the performance of our algorithm when increasing the input size. The collision free position (left) gives great speed-ups: 32, 126, 502 and 1995 times for subdivision levels 0,1,2,3 respectively. The more decent results for the interfering situation (right) are reported in Table 5.3.

are presented in Table 5.3. The speed-up of our algorithm increases with the size of the model. This could have been expected because the proposed on-demand refitting does not depend on the actual geometry of the model, unlike the bottom-up refitting. On the posture from Figure 5.6 (right), we performed also a comparison with rigid body CD. A brand new sphere tree built for the unsubdivided animated dwarves detects collisions in 0.7ms, whereas our algorithm needs 1.33ms. (However, the building of the tree took 1549ms, thus this is of course not a method of choice in practical applications.) It shows that the performance of our deformable CD algorithm is comparable to rigid body CD.

Finally, we tested our CD algorithm on a crowd simulation. The scenario consists of 50 creature models walking in close proximity, see Figure 5.7. In this case, we obtained a 15-times faster collision detection query.

Subdivisions	0	1	2	3
Triangles	1664	6656	26624	106496
Spheres	2046	8821	36506	147025
Collisions	46	90	178	350
On-demand time	1.33	2.17	3.8	8.04
On-demand S-S	4866	9309	17083	37682
Bottom-up time	5.24	18.1	66.37	260
Bottom-up S-S	9515	19833	40527	86437
Speed-up	3.9	8.34	17.47	32.34

Table 5.3: The performance comparison for subdivided dwarf models, Figure 5.6 right. "*Spheres*" is the total number of spheres in the tree and "S-S" stands for the number of sphere-sphere overlap tests. The row "*Speed-up*" is the ratio between average time for CD using the bottom-up refitting and CD with our on-demand refitting.

Figure 5.7: The scenario for our crowd collision detection experiment: 50 animated creatures with more than 650 thousands triangles together. Our CD algorithm needs an average 52.83ms per frame, whereas the CD based on complete BVH refitting takes 779.3ms (15-times more).

6 Collision Detection for Spherical Blend Skinning

The collision detection algorithm described in Chapter 5 is efficient, but limited only to linear blend skinning. This is unfortunate, because of the artifacts of linear blend skinning (see Section 2.2.1). In practical applications, it would be highly desirable to support efficient collision detection (CD) for a skinning method that avoids these artifacts, such as spherical or dual quaternion skinning. However, as apparent from Chapter 5, already the CD for linear blend skinning is not trivial – in spite of the simplicity of the linear blending method.

The CD algorithm for linear blend skinning relies on the fact that linear blending interpolates always within the convex hull of input vertices (see Section 5.2). Since this condition is not true for spherical blend skinning, the adaptation of the algorithm from Chapter 5 is not trivial.

As has been demonstrated in Section 4.4, dual quaternion skinning is even more advantageous than spherical blend skinning. Therefore, one could object that a CD algorithm for dual quaternion skinning would be more desirable than the one for spherical blend skinning. However, the situation is not so simple. Even though dual quaternion blending is simpler to implement, the actual blending mechanism is in fact much more complex than that of spherical blending. This is because the dual quaternion blending operates on a 6-dimensional manifold $\hat{Q}_1$, whereas spherical blending on a 3-dimensional hypersphere S_3. Naturally, it is easier to design bounds in the latter space (because spherical topology, even though non-Euclidean, is still relatively simple and well studied). Therefore, we leave the collision detection for dual quaternion skinning for a future work.

6.1 Problem Decomposition

Our CD algorithm for spherical blend skinning has the same structure as that for linear blend skinning, discussed in Chapter 5. Both are based on a pre-computed hierarchy of bounding spheres. The sphere tree computation is identical in both algorithms – the only difference being in the sphere refitting.

In the following, let us therefore assume that we are refitting a sphere S with center $\mathbf{p}$ and radius r, which encloses some set of triangles. We denote all vertices of those triangles as $\mathbf{v}_1, \ldots, \mathbf{v}_t$. For simplicity of notation, we first assume that all these vertices are influenced by the same joint-set $J = \{j_1, \ldots, j_n\}$, that is $J = J(\mathbf{v}_1) = \ldots J(\mathbf{v}_t)$ (adopting the conventions from Section 5.2). The task is to compute a new sphere which will enclose vertices $\mathbf{v}'_1, \ldots, \mathbf{v}'_t$, computed by Formula (3.7).

The trick to obtaining an algorithm sub-linear in t is to replace the set of vertices $\{\mathbf{v}_1, \ldots \mathbf{v}_t\}$ by the bounding sphere S, which is correct because $\{\mathbf{v}_1, \ldots, \mathbf{v}_t\} \subseteq S$. That is, instead of

bounding $\mathbf{v}'_1, \ldots, \mathbf{v}'_t$ we could bound the set

$$\bigcup_{\mathbf{w} \in W_n} \left(Q_{\mathbf{w}}(S - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c \right) \tag{6.1}$$

Considering the whole sphere S of points instead of only one point is not a big problem, as shown in Section 6.3. A more serious problem is that the set from Formula (6.1) is very conservative, because it disregards the actual vertex weights of $\mathbf{v}_1, \ldots \mathbf{v}_t$. In analogy with Chapter 5, this can be solved by employing the vertex weight bounds $[l_j, h_j]$ for each joint $j \in J$. Recall the definition of the set W'_n of limited convex combinations:

$$W'_n = \{\mathbf{w} \in R^n : l_i \le w_i \le h_i, \ i = 1, \ldots n, \ \sum_{i=1}^{n} w_i = 1\}$$

This set can be applied in Formula (6.1) instead of W_n, yielding tighter bounds. The final set to be bounded by the refitted sphere is therefore

$$\bigcup_{\mathbf{w} \in W'_n} \left(Q_{\mathbf{w}}(S - \mathbf{r}_c) + \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c \right) \tag{6.2}$$

As discussed in Section 5.2, there can be computed points $\mathbf{c}_1, \ldots, \mathbf{c}_m$ (called corners) whose convex hull forms the set W'_n:

$$W'_n = CH(\mathbf{c}_1, \ldots, \mathbf{c}_m) = \left\{ \sum_{k=1}^{m} u_k \mathbf{c}_k : \mathbf{u} \in W_m \right\} \tag{6.3}$$

We derive the bound of the set from Formula (6.2) in three steps. In the first step, we bound the linear component $\sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c$ (Section 6.2) and in the second step, we bound the spherical part $Q_{\mathbf{w}}(S - \mathbf{r}_c)$ (Section 6.3). Finally, both of these bounds are simply added together (Section 6.4) to create the resulting refitted sphere. The final algorithm is presented in Section 6.5.

6.2   Bounding the Linear Part

Unlike the bound of the spherical part, the bound of the linear part of the set from Formula (6.2) can be done in a way similar to Chapter 5 (the situation is even slightly simpler here, because we are now bounding points instead of spheres). This is because the linear part is actually nothing but a linear blending applied to the rotation center $\mathbf{r}_c$, which is a point computed by the spherical blend skinning algorithm (see Section 3.1). Recall an important fact: the rotation center is independent of the vertex weights (it depends only on the actual skinning transformations).

Thanks to the expression of W'_n using corners (Formula (6.3)), we can rewrite the linear part of Formula (6.2) as

$$\left\{ \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c : \mathbf{w} \in W'_n \right\} = \left\{ \sum_{i=1}^{n} \left(\sum_{k=1}^{m} u_k c_{k,i} \right) C_{j_i} \mathbf{r}_c : \mathbf{u} \in W_m \right\}$$

where $c_{k,i}$ denotes i-th component of vector $\mathbf{c}_k$. In the latter term, we can swap the sums, because

$$\sum_{i=1}^{n} \left(\sum_{k=1}^{m} u_k c_{k,i} \right) C_{j_i} \mathbf{r}_c = \sum_{k=1}^{m} u_k \left(\sum_{i=1}^{n} c_{k,i} C_{j_i} \mathbf{r}_c \right)$$

We denote the transformations of the rotation center as:

$$\mathbf{r}'_k = \sum_{i=1}^{n} c_{k,i} C_{j_i} \mathbf{r}_c, \quad k = 1, \ldots, m \tag{6.4}$$

which is correct because $\mathbf{c}_k \in W'_n$ and thus $\sum_{i=1}^{n} c_{k,i} = 1$. Formula (6.4) is actually nothing but linear blending applied to $\mathbf{r}_c$ with weight vector $\mathbf{c}_k$. If we put the equations together, we can write the resulting bound of the linear part as

$$\left\{ \sum_{i=1}^{n} w_i C_{j_i} \mathbf{r}_c : \mathbf{w} \in W'_n \right\} = \left\{ \sum_{k=1}^{m} u_k \mathbf{r}'_k : \mathbf{u} \in W_m \right\} = CH(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$$

We see that the bound of the linear part is just a convex hull of several 3D points. These points are given by the pre-computed corners and Formula (6.4).

6.3   Bounding the Spherical Part

Bounding the spherical part of Formula (6.2) is a little bit more tricky, because we must deal with the quaternion linear blending (QLB) hidden in $Q_{\mathbf{w}}$. Recall that we are refitting sphere S with center $\mathbf{p}$ and radius r, expressed in the reference position. First, we replace the sphere S by its center $\mathbf{p}$:

$$\bigcup_{\mathbf{w} \in W'_n} Q_{\mathbf{w}}(S - \mathbf{r}_c) = \{Q_{\mathbf{w}}(\mathbf{p} - \mathbf{r}_c) : \mathbf{w} \in W'_n\} \oplus \left\{ \mathbf{x} \in R^3 : \|x\| \leq r \right\}$$

where r is the radius of sphere S and $\oplus$ denotes the Minkowski sum. However, the Minkowski sum in the previous equation is actually nothing but a convolution of set $\{Q_{\mathbf{w}}(\mathbf{p} - \mathbf{r}_c) : \mathbf{w} \in W'_n\}$ with a zero center sphere of radius r. In the following, we derive the bounding sphere of $\{Q_{\mathbf{w}}(\mathbf{p} - \mathbf{r}_c) : \mathbf{w} \in W'_n\}$. At the end, we account for the Minkowski sum (convolution) by simply increasing the radius of the resulting sphere by r.

The rest of this section is organized as follows: first, we compute bound on the set of rotations $Q_{\mathbf{w}}$ and express it as a subset of unit quaternion sphere S_3. Second, we apply all

rotations from this set to rotate the vector $\mathbf{p} - \mathbf{r}_c$. The result is some subset of R^3, which is enclosed by a final bounding sphere of the spherical part. The reader should not get confused by the fact that the bounds in both steps will have the same shape (a spherical cap, defined below). The difference is that the bounding of rotations occurs in R^4 (the embedding space of unit quaternions), whereas the bounding of rotated vectors takes place in R^3.

Recall that we denote the quaternions corresponding to the rotational parts of matrices $C_{j_1}, \ldots, C_{j_n}$ by $\mathbf{q}_1, \ldots, \mathbf{q}_n$ (see Section 3.3). Then $Q_\mathbf{w}$ is a rotation matrix given by the quaternion $w_1 \mathbf{q}_1 + \ldots + w_n \mathbf{q}_n$. This is correct, because every non-zero quaternion uniquely determines a 3D rotation (even though not vice-versa). We proceed by constructing a bound of all rotations given by the set of quaternions $\{w_1 \mathbf{q}_1 + \ldots + w_n \mathbf{q}_n : \mathbf{w} \in W_n'\}$. We exploit the fact that QLB applies linear combinations of quaternions, and that quaternions can be interpreted as R^4 vectors. The first step will be therefore similar to that described in Section 6.2, just in R^4 instead of R^3. Using the same corners $\mathbf{c}_1, \ldots, \mathbf{c}_m$ as in Section 6.2, we compute another set of quaternions $\mathbf{q}_1', \ldots, \mathbf{q}_m'$ given by

$$\mathbf{q}_k' = \sum_{i=1}^{n} c_{k,i} \mathbf{q}_i, \quad k = 1, \ldots, m$$

which satisfy the property

$$\{w_1 \mathbf{q}_1 + \ldots + w_n \mathbf{q}_n : \mathbf{w} \in W_n'\} = \{u_1 \mathbf{q}_1' + \ldots + u_m \mathbf{q}_m' : \mathbf{u} \in W_m\}$$

This can also be proven by swapping sums as in Section 6.2. It is therefore sufficient to construct a bound for rotations corresponding to quaternions from $CH(\mathbf{q}_1', \ldots, \mathbf{q}_m')$.

We have chosen only a simple bound of a set of rotations: a spherical cap on S_3 (the sphere of all unit quaternions). Generally, we define a *cap* in any dimension as a non-empty intersection of a sphere surface with a halfspace. In the following, we will also need another expression of cap, defined by the center $\mathbf{s}_s$ of the sphere, point $\mathbf{a}_s$ on the sphere's surface (the cap's apex) and an angle $\alpha_s \in [0, \pi]$. If we denote the radius of the sphere as $r_s = \|\mathbf{a}_s - \mathbf{s}_s\|$, then the cap according to the second definition is given as

$$\left\{ \mathbf{x} \in R^d : \|\mathbf{x} - \mathbf{s}_s\| = r_s, \ \langle \mathbf{x} - \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle \geq r_s^2 \cos(\alpha_s) \right\}$$

It is not difficult to prove that both definitions are equivalent, see the following lemma. An example of a cap is shown in Figure 6.1.

Lemma 6.1. *The two following definitions of a cap of sphere with center $\mathbf{s}_s$ and radius r_s are equivalent:*

(i) A non-empty intersection of a half-space with a surface of sphere with center $\mathbf{s}_s \in R^d$ and radius $r_s \in R$

(ii) $\left\{ \mathbf{x} \in R^d : \|\mathbf{x} - \mathbf{s}_s\| = r_s, \ \langle \mathbf{x} - \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle \geq r_s^2 \cos(\alpha_s) \right\}$, where $\mathbf{a}_s$ is the apex, and α_s the angle.
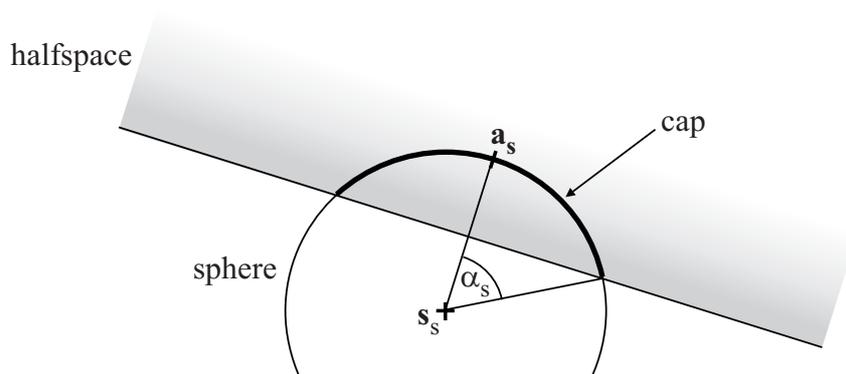
Figure 6.1: Example of a cap in R^2 with center $\mathbf{s}_s$, apex $\mathbf{a}_s$ and angle α_s. In this case the cap is just a spherical arc.

Proof. The half-space from (i) can be written as $\{\mathbf{x} \in R^d : \langle \mathbf{x}, \mathbf{d} \rangle \geq D\}$ for some $\mathbf{d} \in R^d, D \in R$, and the surface of the sphere from (i) as $\{\mathbf{x} \in R^d : \|\mathbf{x} - \mathbf{s}_s\| = r_s\}$. It is therefore sufficient to show that the set (ii) can be written as

$$C = \left\{\mathbf{x} \in R^d : \|\mathbf{x} - \mathbf{s}_s\| = r_s, \ \langle \mathbf{x}, \mathbf{d} \rangle \geq D\right\}$$

The first part is straightforward: if we have a set (ii), we can rewrite $\langle \mathbf{x} - \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle \geq r_s^2 \cos(\alpha_s)$ as $\langle \mathbf{x}, \mathbf{a}_s - \mathbf{s}_s \rangle \geq r_s^2 \cos(\alpha_s) + \langle \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle$. Then it is sufficient to let $\mathbf{d} = \mathbf{a}_s - \mathbf{s}_s$ and $D = r_s^2 \cos(\alpha_s) + \langle \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle$.

The second part requires showing that $\langle \mathbf{x}, \mathbf{d} \rangle \geq D$ can be written as $\langle \mathbf{x}, \mathbf{a}_s - \mathbf{s}_s \rangle \geq r_s^2 \cos(\alpha_s) + \langle \mathbf{s}_s, \mathbf{a}_s - \mathbf{s}_s \rangle$ for some $\mathbf{a}_s \in R^d, \alpha_s \in R$. Without loss of generality, we can assume that $\|\mathbf{d}\| = r_s$ (because $\langle \mathbf{x}, \mathbf{d} \rangle \geq D$ can be multiplied by any non-negative scalar and still represents the same half-space). The apex is then given simply as $\mathbf{a}_s = \mathbf{d} + \mathbf{s}_s$. It remains to find an angle α_s satisfying equation $D = r_s^2 \cos(\alpha_s) + \langle \mathbf{s}_s, \mathbf{d} \rangle$. To complete the proof, it is thus sufficient to verify that $|D - \langle \mathbf{s}_s, \mathbf{d} \rangle| \leq r_s^2$ (so that $\cos(\alpha_s)$ is properly defined by the equation $D = r_s^2 \cos(\alpha_s) + \langle \mathbf{s}_s, \mathbf{d} \rangle$). To show this, we use the fact that the half-space $\langle \mathbf{x}, \mathbf{d} \rangle \geq D$ intersects the spherical surface (here we use the non-emptiness of the intersection in definition (i)). This means that the distance from plane $\langle \mathbf{x}, \mathbf{d} \rangle = D$ to center $\mathbf{s}_s$ is less than or equal to r_s. The distance from $\langle \mathbf{x}, \mathbf{d} \rangle = D$ to $\mathbf{s}_s$ is given by the formula $\frac{|D - \langle \mathbf{s}_s, \mathbf{d} \rangle|}{r_s}$, so the previous condition can be written as

$$\frac{|D - \langle \mathbf{s}_s, \mathbf{d} \rangle|}{r_s} \leq r_s \ \Rightarrow \ |D - \langle \mathbf{s}_s, \mathbf{d} \rangle| \leq r_s^2$$

as we wanted to prove. $\square$

The bound of a set of rotations expressed by a cap C on S_3 has a nice geometric interpretation. Let us denote the apex of cap C as $\mathbf{a}_C$ and the angle as α_C (in this case, the center $\mathbf{s}_C = \mathbf{0}$ and radius $r_C = 1$, because S_3 is a zero centered sphere with unit radius). Since we

are considering S_3, the apex $\mathbf{a}_C$ is a unit quaternion representing some rotation R_C. Then all rotations represented by cap C can be obtained by composing R_C with a rotation about an arbitrary axis and angle within $[0, 2\alpha_C]$ ($2\alpha_C$ because quaternions work with *half* of the angle of rotation, see Section 2.1.1). If we have the set of rotations bound by cap $C \subseteq S_3$, we can bound our original set $\{Q_\mathbf{w}(\mathbf{p} - \mathbf{r}_c) : \mathbf{w} \in W'_n\}$ by $\rho = \{R(\mathbf{p} - \mathbf{r}_c) : R \in C\}$, where C is interpreted as a set of rotations. But the set ρ is nothing but the set of all possible rotations of vector $R_C(\mathbf{p} - \mathbf{r}_c)$ along an arbitrary axis and angle within $[0, 2\alpha_C]$. It means that ρ is nothing but another spherical cap (but now in R^3)! The apex of cap ρ is $R_C(\mathbf{p} - \mathbf{r}_c)$, center is $\mathbf{0}$ and angle $2\alpha_C$.

Since we are constructing a cap on S_3, we normalize our quaternions $\mathbf{q}'_1, \ldots, \mathbf{q}'_m$ to unit quaternions: $\mathbf{q}''_k = \mathbf{q}'_k / \|\mathbf{q}'_k\|$, $k = 1, \ldots, m$. We can work with $\mathbf{q}''_k$ instead of $\mathbf{q}'_k$, because the following lemma shows that the sets of rotations corresponding to $CH(\mathbf{q}'_1, \ldots, \mathbf{q}'_m)$ and $CH(\mathbf{q}''_1, \ldots, \mathbf{q}''_m)$ are the same.

Lemma 6.2. *Let $\mathbf{q}_1, \ldots, \mathbf{q}_m$ be non-zero quaternions and $\mathbf{n}_1 = \frac{\mathbf{q}_1}{\|\mathbf{q}_1\|}, \ldots, \mathbf{n}_m = \frac{\mathbf{q}_m}{\|\mathbf{q}_m\|}$ their corresponding unit quaternions. Then the set $M = CH(\mathbf{q}_1, \ldots, \mathbf{q}_m)$ represents the same rotations as the set $N = CH(\mathbf{n}_1, \ldots, \mathbf{n}_m)$.*

Proof. We know that two non-zero quaternions $\mathbf{p}, \mathbf{q}$ represent the same rotation if and only if there exists $k \in R$, $k \neq 0$ such that $\mathbf{p} = k\mathbf{q}$. First, we show that any rotation from M is also present in N. Let us choose an arbitrary $\mathbf{a} \in M$, i.e., $\mathbf{a} = \sum w_i \mathbf{q}_i$ for some $\mathbf{w} \in W_m$. We define $K = \sum w_i \|\mathbf{q}_i\|$ and $u_i = \frac{w_i \|\mathbf{q}_i\|}{K}$. Obviously $K > 0$, $u_i \geq 0$ and $\sum u_i = 1$, that is $\mathbf{u} \in W_m$. Hence $\sum u_i \mathbf{n}_i \in N$ and to finish the first part of the proof it is sufficient to show that $K \cdot \sum u_i \mathbf{n}_i = \mathbf{a}$ (i.e. that $\sum u_i \mathbf{n}_i$ represents the same rotation as $\mathbf{a}$). So, we show that:

$$K \cdot \sum u_i \mathbf{n}_i = K \cdot \sum \frac{w_i \|\mathbf{q}_i\| \mathbf{q}_i}{K \|\mathbf{q}_i\|} = \sum w_i \mathbf{q}_i = \mathbf{a}$$

Second, we show that any rotation from N is also present in M. Let us choose an arbitrary $\mathbf{b} \in N$, i.e. $\mathbf{b} = \sum t_i \mathbf{n}_i$ for some $\mathbf{t} \in W_m$. We define $L = \sum \frac{t_i}{\|\mathbf{q}_i\|}$ and $s_i = \frac{t_i}{\|\mathbf{q}_i\| L}$. Again, $L > 0$ and $\mathbf{s} \in W_m$, therefore $\sum s_i \mathbf{q}_i \in M$. In a similar way as before we see that

$$L \cdot \sum s_i \mathbf{q}_i = L \cdot \sum \frac{t_i \mathbf{q}_i}{\|\mathbf{q}_i\| L} = \sum t_i \mathbf{n}_i = \mathbf{b}$$

which completes the proof. □

Now, it remains just to construct a cap on S_3 containing our quaternions $\mathbf{q}''_1, \ldots, \mathbf{q}''_m$. In order to construct this cap, we bound $\mathbf{q}''_1, \ldots, \mathbf{q}''_m$ by an enclosing sphere $E \subseteq R^4$ with center $\mathbf{s}_E$ and radius r_E. We can assume that the radius of sphere E satisfies $r_E < 1$: if not, we can simply consider the whole S_3 (of radius 1) which bounds all rotations, as the bounding cap (although it should be noted that this situation never occurred during practical experiments).

We compute this sphere using Algorithm 5.1 (considering points as zero radius spheres). At this stage, we are almost done, because $E \cap S_3$ is the desired cap. This is illustrated in Figure 6.2 and verified in the following two lemmas.
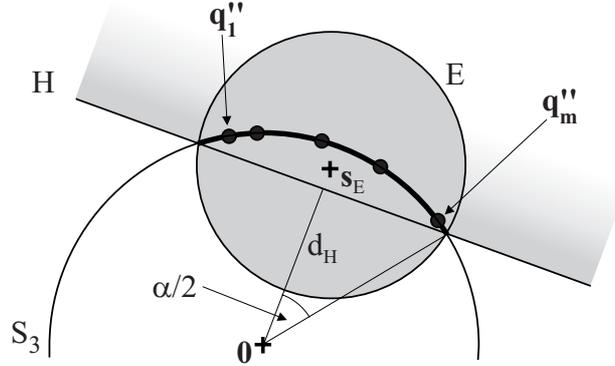
Figure 6.2: To construct the bounding cap (in this picture just the spherical arc) for quaternions $\mathbf{q}''_1, \ldots, \mathbf{q}''_m$, we first create an enclosing sphere E (not necessarily the smallest one). The cap is then given as $E \cap S_3$, which can be equally expressed as $H \cap S_3$, where H is a halfspace from Lemma 6.3. A formula for distance d_H from H to $\mathbf{0}$ is also derived in Lemma 6.3. The distance d_H is used to compute the angle α.

Lemma 6.3. *Let S_a be a spherical surface in R^d with center $\mathbf{a}$ and radius r_a. Let S_b be a sphere in R^d with center $\mathbf{b}$ and radius r_b. Then the intersection $S_a \cap S_b$ is a cap, i.e. $S_a \cap S_b = S_a \cap H$, where H is a halfspace in R^d. Moreover, if $\mathbf{a} = \mathbf{0}, r_a = 1$ and $r_b < 1$, then $\mathbf{0} \notin H$ and the distance from $\mathbf{0}$ to H is $\frac{1+\|\mathbf{b}\|^2 - r_b^2}{2\|\mathbf{b}\|}$.*

Proof. The sets S_a and S_b can be written as

$$S_a = \left\{\mathbf{x} \in R^d : \sum (x_i - a_i)^2 = r_a^2\right\}$$
$$S_b = \left\{\mathbf{x} \in R^d : \sum (x_i - b_i)^2 \leq r_b^2\right\}$$

Therefore the intersection $S_a \cap S_b = \left\{\mathbf{x} \in R^d : \sum(x_i - a_i)^2 = r_a^2, \ \sum(x_i - b_i)^2 \leq r_b^2\right\}$. The system of these two formulas can be written as

$$\sum(x_i^2 - 2x_i a_i + a_i^2) \;=\; r_a^2 \tag{6.5}$$
$$\sum(x_i^2 - 2x_i b_i + b_i^2) \;\leq\; r_b^2 \tag{6.6}$$

which is equivalent to the system

$$\sum(x_i - a_i)^2 \;=\; r_a^2 \tag{6.7}$$
$$\sum(2(a_i - b_i)x_i + b_i^2 - a_i^2) \;\leq\; r_b^2 - r_a^2 \tag{6.8}$$

because Formula (6.8) is simply Formula (6.6) minus Formula (6.5). However, Formula (6.8) is an equation describing a halfspace, which we can denote as H. This proves the first part of the statement. To show the second part, we substitute $\mathbf{x} = \mathbf{0}$, $\mathbf{a} = 0$, $r_a = 1$ into Formula (6.8) and we obtain $\sum b_i^2 \leq r_b^2 - 1$. Since we supposed $r_b < 1$, this equation obviously cannot be satisfied, which means that $\mathbf{x} = \mathbf{0}$ cannot be in H. Therefore, the distance from H to $\mathbf{0}$ is the same as the distance from the hyperplane determining H to $\mathbf{0}$. Generally, the distance of hyperplane $\langle \mathbf{x}, \mathbf{d} \rangle = D$, $\mathbf{d} \in R^d$, $D \in R$ from $\mathbf{0}$ is $|D|/\|\mathbf{d}\|$. In our case, we have $|D| = |r_b^2 - 1 - \|\mathbf{b}\|^2| = 1 + \|\mathbf{b}\|^2 - r_b^2$ and $\|\mathbf{d}\| = 2\|\mathbf{b}\|$, which proves the last part of the statement. $\qquad\square$

Lemma 6.4. *Let $\mathbf{n}_1, \ldots, \mathbf{n}_m$ be unit quaternions enclosed by sphere $E \subseteq R^4$ with radius < 1. Then the set $C = E \cap S_3$ is a cap such that*

$$\forall \mathbf{w} \in W_m : \quad \frac{w_1 \mathbf{n}_1 + \ldots + w_m \mathbf{n}_m}{\|w_1 \mathbf{n}_1 + \ldots + w_m \mathbf{n}_m\|} \in C$$

Proof. From Lemma 6.3 we know that C is really a cap and can be written as $C = H \cap S_3$, where H is some halfspace not containing the zero vector. Obviously $\mathbf{n}_i \in C$ (because $\mathbf{n}_i \in S_3$ and $\mathbf{n}_i \in E$), therefore it must be also true that $\mathbf{n}_i \in H$. Since a halfspace is always convex, we have $w_1 \mathbf{n}_1 + \ldots + w_m \mathbf{n}_m \in H$. We denote $\mathbf{n}' = w_1 \mathbf{n}_1 + \ldots + w_m \mathbf{n}_m$. Since obviously $\mathbf{n}'/\|\mathbf{n}'\| \in S_3$, it remains to show only that $\mathbf{n}'/\|\mathbf{n}'\| \in H$. First, we apply the triangle inequality to obtain

$$\|\mathbf{n}'\| = \left\| \sum w_i \mathbf{n}_i \right\| \leq \sum \|w_i \mathbf{n}_i\| = \sum w_i \|\mathbf{n}_i\| = \sum w_i = 1$$

that is $1/\|\mathbf{n}'\| \geq 1$. Second, we show that $\gamma \mathbf{n}' \in H$ for any $\gamma \geq 1$, especially for $\gamma = 1/\|\mathbf{n}'\|$. Since $\mathbf{0} \notin H$, the halfspace H can be expressed as $H = \{\mathbf{x} \in R^4 : \langle \mathbf{x}, \mathbf{d} \rangle \geq 1\}$ for some vector $\mathbf{d} \in R^4$. We know that $\mathbf{n}' \in H$, which means that $\langle \mathbf{n}', \mathbf{d} \rangle \geq 1$ and therefore also $\langle \gamma \mathbf{n}', \mathbf{d} \rangle \geq 1$, i.e. $\gamma \mathbf{n}' \in H$, which is what we wanted to prove. $\qquad\square$

Corollary. *Let $\mathbf{n}_1, \ldots, \mathbf{n}_m$ be unit quaternions and C cap as above. Then all rotations represented by $CH(\mathbf{n}_1, \ldots, \mathbf{n}_m)$ are also present in C.*

Now it is straightforward to derive the resulting cap $C' \subseteq R^3$ which fulfills

$$\{Q_{\mathbf{w}}(\mathbf{p} - \mathbf{r}_c) : \mathbf{w} \in W_n'\} \subseteq C'$$

We denote by Q_c the rotation corresponding to $\mathbf{s}_E$. The apex of the cap C' is then $Q_c(\mathbf{p} - \mathbf{r}_c)$, the center is $\mathbf{0}$ and the angle is

$$\alpha = 2\arccos(d_H), \ d_H = \frac{1 + \|\mathbf{s}_E\|^2 - r_E^2}{2\|\mathbf{s}_E\|} \tag{6.9}$$

where d_H denotes the distance from H to $\mathbf{0}$, as computed in Lemma 6.3 and illustrated in Figure 6.2. Note that, since the sphere E intersects S_3, it cannot happen that E is

strictly inside S_3, i.e., $\|\mathbf{s}_E\| + r_E < 1$ cannot be true. Therefore, $\|\mathbf{s}_E\| + r_E \geq 1$, thus $-r_E^2 \leq -(1 - \|\mathbf{s}_E\|)^2 = -1 + 2\|\mathbf{s}_E\| - \|\mathbf{s}_E\|^2$ from which follows:

$$d_H \leq \frac{1 + \|\mathbf{s}_E\|^2 - 1 + 2\|\mathbf{s}_E\| - \|\mathbf{s}_E\|^2}{2\|\mathbf{s}_E\|} = \frac{2\|\mathbf{s}_E\|}{2\|\mathbf{s}_E\|} = 1$$

Since obviously $0 \leq d_H$, the arccos in Formula (6.9) is well defined and $\alpha \in [0, \pi]$.

The resulting sphere returned for the bound of the spherical part is nothing but a minimal enclosing sphere of cap C'. We denote this minimal enclosing sphere as $F \subseteq R^3$ and we compute it easily: its center is $\cos(\alpha) Q_c(\mathbf{p} - \mathbf{r}_c)$ and radius is $\sin(\alpha)\|Q_c(\mathbf{p} - \mathbf{r}_c)\| = \sin(\alpha)\|\mathbf{p} - \mathbf{r}_c\|$, where α is given by Formula (6.9). This is illustrated in Figure 6.3 and proven in the following lemma.

Lemma 6.5. *Let $C \subseteq R^d$ be a cap with center $\mathbf{0}$, radius r, apex $\mathbf{a}$ and angle $\alpha \in [0, \pi]$. Then the smallest enclosing sphere S of cap C has center $\mathbf{a}\cos\alpha$ and radius $\|\mathbf{a}\|\sin\alpha$.*

Proof. First, we show that cap C cannot be enclosed by a sphere with smaller radius than $\|\mathbf{a}\|\sin\alpha$. This is because there exist two vectors $\mathbf{v}_1, \mathbf{v}_2 \in C$ whose distance is $2\|\mathbf{a}\|\sin\alpha$. To construct those vectors, pick an arbitrary vector $\mathbf{v}$ such that $\langle \mathbf{v}, \mathbf{a} \rangle = 0$ and $\|\mathbf{v}\| = \|\mathbf{a}\| = r$. Now we can define $\mathbf{v}_1 = \mathbf{a}\cos\alpha + \mathbf{v}\sin\alpha$ and $\mathbf{v}_2 = \mathbf{a}\cos\alpha - \mathbf{v}\sin\alpha$. Their distance is obviously $2\|\mathbf{v}\|\sin\alpha = 2\|\mathbf{a}\|\sin\alpha$, so it remains to verify that really $\mathbf{v}_1, \mathbf{v}_2 \in C$. We compute that

$$\|\mathbf{v}_1\| = \sqrt{\|\mathbf{a}\|^2\cos^2\alpha + \|\mathbf{v}\|^2\sin^2\alpha} = \sqrt{r^2(\cos^2\alpha + \sin^2\alpha)} = r$$

and

$$\langle \mathbf{v}_1, \mathbf{a} \rangle = \langle \mathbf{a}\cos\alpha + \mathbf{v}\sin\alpha, \mathbf{a} \rangle = \langle \mathbf{a}, \mathbf{a} \rangle \cos\alpha + \langle \mathbf{v}, \mathbf{a} \rangle \sin\alpha = r^2\cos\alpha$$

which shows that $\mathbf{v}_1 \in C$. The same reasoning can be repeated for $\mathbf{v}_2$, showing that also $\mathbf{v}_2 \in C$.

In the rest of the proof, we have to verify that $C \subseteq S$. Therefore, let us pick an arbitrary $\mathbf{x} \in C$. According to the definition of the cap, it means that $\|\mathbf{x}\| = r$ and $\langle \mathbf{x}, \mathbf{a} \rangle \geq r^2\cos\alpha$. To show that $\mathbf{x} \in S$, we compute

$$\|\mathbf{x} - \mathbf{a}\cos\alpha\|^2 = \langle \mathbf{x} - \mathbf{a}\cos\alpha, \mathbf{x} - \mathbf{a}\cos\alpha \rangle = \|\mathbf{x}\|^2 - 2\langle \mathbf{x}, \mathbf{a}\cos\alpha \rangle + \cos^2\alpha\|\mathbf{a}\|^2$$

Now we use the fact that $\|\mathbf{a}\| = \|\mathbf{x}\| = r$ and $-2\langle \mathbf{x}, \mathbf{a}\cos\alpha \rangle \leq -2r^2\cos^2\alpha$ to obtain

$$\|\mathbf{x}\|^2 - 2\langle \mathbf{x}, \mathbf{a}\cos\alpha \rangle + \cos^2\alpha\|\mathbf{a}\|^2 \leq r^2 - 2r^2\cos^2\alpha + r^2\cos^2\alpha = r^2(1 - \cos^2\alpha) = r^2\sin^2\alpha$$

Taking the square root on both sides yields $\|\mathbf{x} - \mathbf{a}\cos\alpha\| \leq r\sin\alpha$, that is $\mathbf{x} \in S$. $\qquad\square$

Final note: it may seem that the bound of the spherical part could be done in an easier way than by constructing a cap on S_3. In fact, we have also considered other more simple approaches, but none of them worked in general. For example, it is not correct to just rotate vector $\mathbf{p} - \mathbf{r}_c$ by quaternions $\mathbf{q}_1'', \ldots, \mathbf{q}_m''$ and bound the results directly by a 3D enclosing sphere. However, the algorithm presented in this section is only difficult to explain and justify – the actual implementation is fairly simple, see Section 6.5.
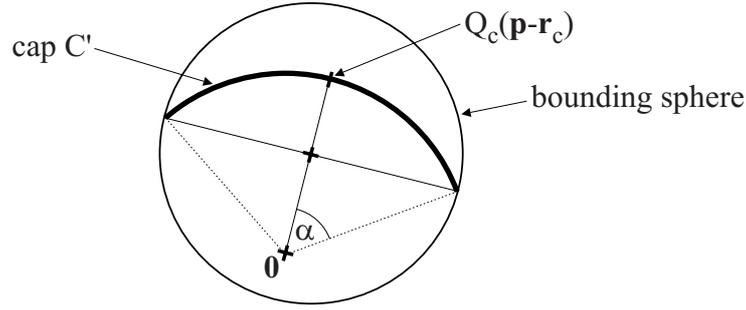
Figure 6.3: If the cap C' centered in the origin has apex $Q_c(\mathbf{p} - \mathbf{r}_c)$ and angle α, then its minimal enclosing sphere has center $\cos(\alpha)Q_c(\mathbf{p} - \mathbf{r}_c)$ and radius $\sin(\alpha)\|\mathbf{p} - \mathbf{r}_c\|$.

6.4   Putting the Bounds Together

To construct the final bounding sphere of the set from Formula (6.2), it remains just to combine the bound of the linear part and the bound of the spherical part. Recall that, in Section 6.2, the bound of the linear part was expressed as $CH(\mathbf{r}'_1, \ldots, \mathbf{r}'_m)$ and, in Section 6.3, the spherical part was bound by sphere F. The bound of both parts can therefore be expressed as $CH(\mathbf{r}'_1 \oplus F, \ldots, \mathbf{r}'_m \oplus F)$, i.e., the final bounding sphere encloses $\mathbf{r}'_1 \oplus F, \ldots, \mathbf{r}'_m \oplus F$. This enclosing sphere is again computed by Algorithm 5.1.

In the beginning of this chapter, we assumed that all vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$ of the reference sphere S are assigned to only one joint-set J. If this is not the case, i.e., the vertices $\mathbf{v}_1, \ldots, \mathbf{v}_t$ are influenced by more joint-sets $J_1, \ldots, J_z$, we simply repeat the same algorithm for each of these joint-sets. This way, we obtain spheres $\mathbf{r}'_{1,1} \oplus F_1, \ldots, \mathbf{r}'_{1,m_1} \oplus F_1, \mathbf{r}'_{2,1} \oplus F_2, \ldots, \mathbf{r}'_{2,m_2} \oplus F_2, \ldots, \mathbf{r}'_{z,1} \oplus F_z, \ldots, \mathbf{r}'_{z,m_z} \oplus F_z$ and enclose them by one bounding sphere as before.

6.5   The Final Algorithm

This section presents the final sphere refitting, Algorithm 6.1. For brevity, we write $(\mathbf{s}, r)$ to denote a data structure describing sphere with center $\mathbf{s}$ and radius r. List L_2 actually stores points $\mathbf{q}''_h$ but for simplicity, we treat them as spheres with zero radius, i.e.: $(\mathbf{q}''_h, 0)$.

The addition of $(\mathbf{0}, r)$ on line (14) simply inflates the radius of the resulting sphere by r. The sphere refitting algorithm uses a subroutine QUAT2MATRIX to convert from a quaternion to matrix representation. The pseudocode of Algorithm 6.1 is written for clarity and not for maximal performance – in the actual implementation, we perform some optimization tricks. For example, when a joint-set consists of only one joint (which is a typical situation on lower levels of the tree), it is of course not necessary to compute any bounds – the sphere can be transformed directly.

Algorithm 6.1: Sphere refitting for spherical blend skinning

Input: $S = (\mathbf{p}, r)$ – sphere to be refitted (expressed in the reference posture)

$C_1, \ldots, C_p$ – joint transformation matrices for the current posture

$\mathbf{q}_1, \ldots, \mathbf{q}_p$ – quaternions corresponding to $C_1, \ldots, C_p$

$J_1, \ldots, J_z$ – joint-sets influencing sphere S

$\mathbf{c}_{1,1}, \ldots, \mathbf{c}_{1,m_1}, \ldots, \mathbf{c}_{z,1}, \ldots, \mathbf{c}_{z,m_z}$ – pre-computed corners

$\mathbf{r}_c$ – rotation center computed by spherical blend skinning

Output: sphere S refitted for current skin deformation

SBSSPHEREREFIT(S)

(1)　　$L_1 =$ empty list

(2)　　**for** $i = 1$ **to** z

(3)　　　$L_2 =$ empty list

(4)　　　**for** $h = 1$ **to** m_i

(5)　　　　$\mathbf{r}'_h = \sum_{g=1}^{n} c_{i,h,g} C_{j_g} \mathbf{r}_c$

(6)　　　　$\mathbf{q}'_h = \sum_{g=1}^{n} c_{i,h,g} \mathbf{q}_g$

(7)　　　　$\mathbf{q}''_h = \mathbf{q}'_h / \|\mathbf{q}'_h\|$

(8)　　　　insert sphere $(\mathbf{q}''_h, 0)$ into list L_2

(9)　　　$(\mathbf{c}_E, r_E) =$ BOUNDINGSPHERE(L_2)

(10)　　　$Q_c =$ QUAT2MATRIX$(\mathbf{c}_E / \|\mathbf{c}_E\|)$

(11)　　　$\alpha = 2 \arccos(\frac{1 + \|\mathbf{c}_E\|^2 - r_E^2}{2\|\mathbf{c}_E\|})$

(12)　　　**for** $h = 1$ **to** m_i

(13)　　　　insert sphere $(\mathbf{r}'_h + \cos(\alpha) Q_c(\mathbf{p} - \mathbf{r}_c), \sin(\alpha)\|\mathbf{p} - \mathbf{r}_c\|)$ into list L_1

(14)　　**return** BOUNDINGSPHERE$(L_1) + (\mathbf{0}, r)$

6.6　Results

In order to provide comparative measurements, we execute the tests on the same models and animations as in Chapter 5 (the man model with 4435 vertices, 8270 triangles and 27 joints, and the creature model with 6682 vertices, 13590 triangles and 56 joints). The dwarf model from Chapter 5 has only rigid skinning (trivial vertex weights), and thus there is no reason to apply spherical blending. First, we investigate the tightness of the refitted spheres, see Figure 6.4 and Table 6.1.

We observe that the size of the refitted spheres is almost the same as the size of the spheres in the reference posture, which are optimal because they are computed by an exact minimal enclosing sphere algorithm [36]. Also, the size of the spheres refitted for linear and spherical blending is similar – even though the geometries of the deformed skins are different (observe the candy wrapper artifact in the middle row of Figure 6.4). The average sphere radii are reported in Table 6.1.

We measured the speed of the collision detection and sphere refitting on a 2.5GHz Athlon PC under normal working conditions. Please note that the comparison of timings of
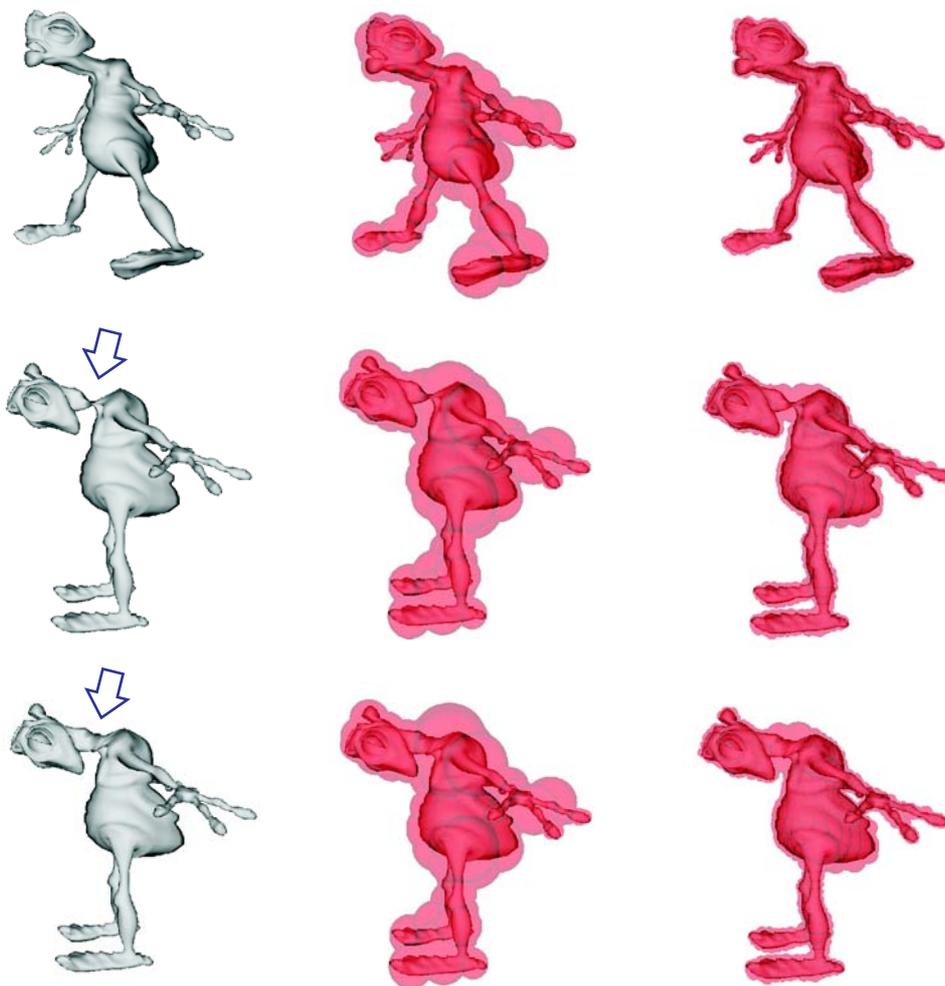
Figure 6.4: Bounding spheres on levels 4 and 6 of the tree: **Top:** reference posture, minimal enclosing spheres. **Middle:** animated posture deformed with linear blend skinning (note the candy wrapper artifact in the neck), spheres refitted by Algorithm 5.2. **Bottom:** the same posture as in the middle row deformed by spherical blend skinning, spheres refitted by Algorithm 6.1.

sphere refitting for linear and spherical blending cannot be exact, because both skinning methods produce slightly different geometry (and thus possibly also a different number of intersections). However, since the test animations involve only moderate joint rotations, the difference between linear and spherical skinning is not very big (the difference is obvious only for large joint rotations, e.g., the neck twist in Figure 6.4). The first scenario is an animation of two walking men (the same as in Chapter 5).

Results for this animation are reported in the first row of Table 6.2. We see that the slowdown caused by an advanced skinning algorithm is really negligible. The refitting of all 15339 spheres requires a total time of 20.15ms, which is 1.31μs per sphere. This is almost

Level	Reference	Linear Blending	Spherical Blending	Best
1	34.55	72.17	79.55	34.41
2	18.86	31.58	33.95	22.05
3	7.38	8.40	8.89	7.39
4	3.68	4.03	4.12	3.69
5	1.69	1.80	1.83	1.70
6	0.91	0.97	0.98	0.92
7	0.61	0.65	0.66	0.61
8	0.42	0.43	0.43	0.42
9	0.30	0.30	0.30	0.30

Table 6.1: This table lists an average radii of spheres in the creature model's sphere tree. **Reference:** spheres for the reference posture (Figure 6.4 top). **Linear Blending:** animated posture in Figure 6.4 middle, skin deformed by linear blending, spheres refitted by Algorithm 5.2 from Chapter 5. **Spherical Blending:** animated posture in Figure 6.4 bottom, skin deformed by spherical blending, spheres refitted by Algorithm 6.1. **Best:** as in Spherical Blending, but spheres refitted by an exact minimal enclosing sphere algorithm [36].

as good as sphere refitting for linear blending (which requires about 1μs per sphere, see Section 5.4). In practical situations, of course, only a small fraction of all those spheres is refitted, therefore times for collision detection are much smaller, see Table 6.2.

Scenario	Linear Blending	Spherical Blending
Men (Full)	0.27	0.31
Creatures (Full)	6.14	7.47
Creatures (Yes/no)	0.72	1.44

Table 6.2: Average times in milliseconds for one collision detection query in various settings. **Full:** CD returns set of all colliding triangles, **Yes/no:** CD returns only one pair of colliding triangles, if any. **Linear Blending:** on-demand refitting for linear blending (Algorithm 5.2). **Spherical Blending:** on-demand refitting for spherical blending (Algorithm 5.2). We see that the performance of on-demand refitting for spherical blending is comparable to that for linear blending.

The next testing scenario is the torture test, because of the many colliding triangles (much more than in practical situations, where the collision response routines prevent such extreme interpenetration). Again, the only difference from the experiment in Chapter 5 is in the application of spherical blend skinning instead of linear. In this animation, we measured besides the standard full CD query also the average time for yes/no CD task, which reports only whether the objects are colliding or not, without searching for all colliding triangles (collision detection algorithm in this case stops when the first intersecting triangle pair is found). Measurements are reported in the second and third row of Table 6.2. We
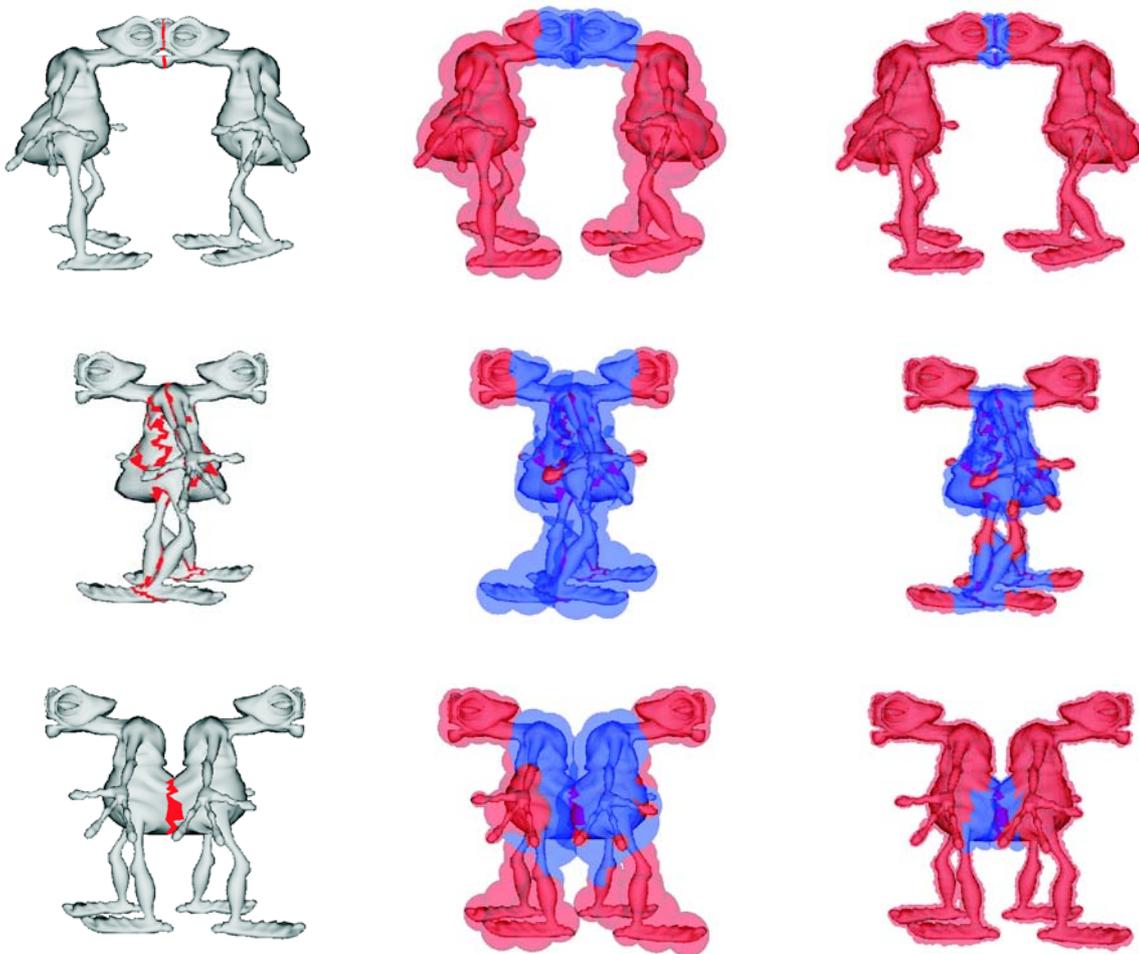
Figure 6.5: A "walk-through" animation, involving a lot of collisions. In the second and third column we see spheres on level 4 and 6 refitted by our method. This scenario demonstrates that our algorithm is suitable even in difficult situations.

see that, even in difficult situations, the overhead for spherical skinning is fortunately very low, and thus its performance is comparable to refitting for linear blend skinning.

7 Skinning Arbitrary Deformations

Previous chapters consider the situation when our 3D objects are already rigged and animated (i.e., equipped with skinning transformations and vertex weights, as discussed in Section 2.2). This is usually done manually with the aid of professional software [9, 10]. However, manual rigging and animation is not always convenient, especially when we already have an animation computed by other means (e.g., physically based simulation). This was one of the motivations for James and Twigg's Skinned Mesh Animations (SMAs), which is a method to automatically infer the rigging structure and skinning transformations from a pre-computed animation [63].

Their method works by clustering 3D model to quasi-rigid components (i.e., approximately rigid parts). The transformations of quasi-rigid components are computed directly from their actual positions and orientations in the animation. Vertex weights are optimized in a way similar to [99]. This process constructs a skinned approximation of the original animation. Skinned animations offer interesting data reduction when compared to the classic way of handling pre-computed animations (i.e., storing the positions of animated vertices for every keyframe). Therefore, automatically constructed skinning approximations can be considered as an animation compression method. However, their only benefit is not data reduction: SMAs also offer efficient hardware-accelerated rendering, rest-pose editing and fast collision detection [63].

However, James and Twigg's method works efficiently only for quasi-articulated 3D objects (i.e., objects created by linking quasi-rigid components). Unfortunately, SMAs do not perform as well for highly deformable animations, such as those of cloth or elastic materials. This is understandable because in such animations, no quasi-rigid components can be found. Does it mean that no efficient skinning approximation exists in this case? In fact, experiences with animating dressed virtual humans indicate the contrary – see Section 1.4.

As an attempt to answer this question in general, we propose a new method to automatically construct skinned approximations. Since we do not make any assumptions about the input animation, we distribute the control transformations (called *proxy-joints*, in analogy to SMAs) uniformly over the rest-pose mesh. Subsequently, for each frame of the animation, our algorithm finds the set of transformations whose application in matrix palette skinning approximates the current shape of the model as closely as possible (see Figure 7.1). This way, we ensure that the overall deformation is approximated reasonably, without relying on quasi-rigid components.

Of course, this does not come for free: our pre-computation times are slower than those of SMAs and our algorithm does not find the smallest reasonable number of proxy-joints automatically (as a result, we typically use more proxy-joints than necessary). On the other hand, according to our experiments, our algorithm constructs much more accurate skinning approximations of highly deformable 3D models than SMAs. This enables us to exploit the advantages of skinned animations even outside the realm of quasi-articulated models (e.g., for cloth and elastic materials).
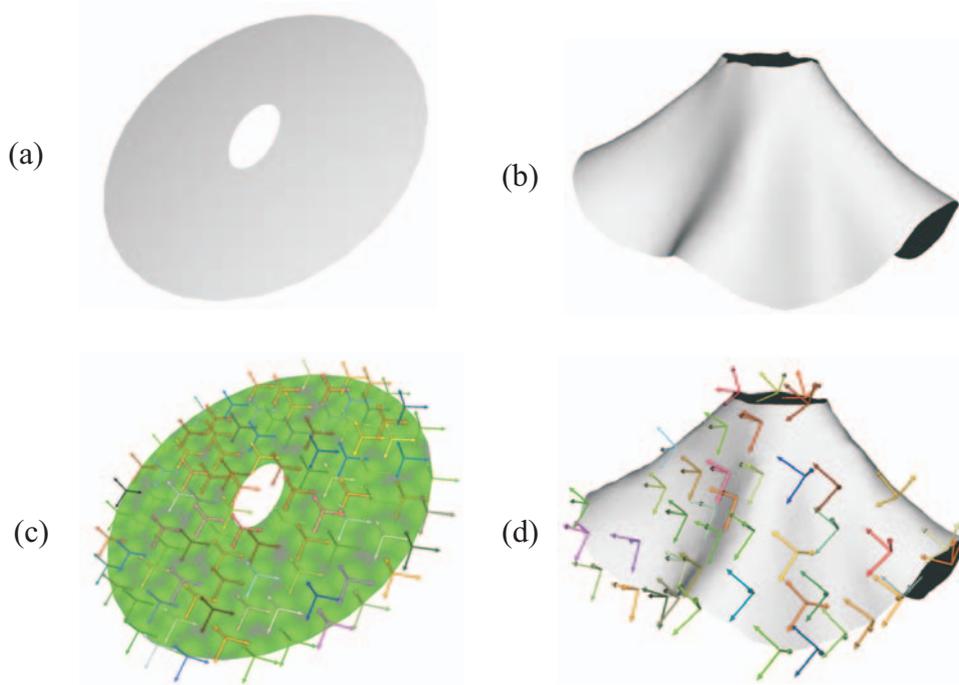
Figure 7.1: Overview of our method for skinning arbitrary deformations: as input we have the rest-pose model (a), and a deformed one (b). Our algorithm first determines the proxy-joints and their influences (c) and then computes the joint transformations, whose application in matrix palette skinning (d) gives a good approximation of the input deformation. Even though (b) and (d) appear to be almost identical, (d) needs about 17 times less memory than (b) and can be rendered efficiently as a skinned mesh.

As input we have a sequence of polygonal meshes with constant connectivity. Furthermore, we assume that a rest-pose mesh is given. This can be as simple as the first mesh of the animation. If available, we can also use the rest-pose of the mesh used to produce the animation, e.g., an unfolded piece of cloth. In order to support more accurate shading, vertex normals are usually stored for each keyframe also. If the model is already rigged, i.e., equipped with joints and their influences, we can skip the following section and proceed directly to transformation fitting (Section 7.2). However, in general, we do not assume that those structures are present (as is the case for our experimental data). Therefore, an algorithm for their automatic generation is described below.

7.1   Automatic Rigging

The proxy-joints are distributed over the rest-pose mesh so that each proxy-joint influences approximately the same amount of geometry. Let us assume that we want to use p proxy-joints (samples). We need to position the proxy joints in space so that the maximal distance

of a rest-pose vertex to the nearest proxy-joint is minimized. This ensures that each proxy-joint controls approximately the same amount of geometry, even if the vertices of the rest-pose mesh are non-uniformly distributed. This problem is known in computational geometry as the p-center problem [1]. The simple greedy algorithm has been shown to produce good results [38]. This algorithm places the centers (in our case, the proxy-joints) so that they coincide with the mesh vertices. At each step of the algorithm, a new proxy-joint is created at the vertex that has maximal distance from all proxy-joints created so far. The resulting sampling is illustrated in Figure 7.2. Note that in our method (as well is in SMAs [63]), every proxy-joint is independent, and there is no hierarchical structure as in a skeleton.

The joint influences (vertex weights) are also easily computed. Let r be the maximal distance from a rest-pose mesh vertex to the nearest joint (i.e., the value that the p-center solution minimizes). The influence of each joint is limited to the ball centered in the joint with radius $P \cdot r$, where P is a user-defined parameter controlling the area of the joint's influence. We found the value $P = 1.5$ to perform well in practice. The weight decays linearly from 1 at the joint's center to 0 on the ball's boundary. If more than one joint influences the vertex, as is usually the case, we normalize the weights of all joints so that they sum to 1. Since weights are obviously non-negative, this guarantees convex vertex weights. We also experimented with a more sophisticated weight assignment, such as one based on least-squares optimization [99], but we found the improvement to be almost negligible.
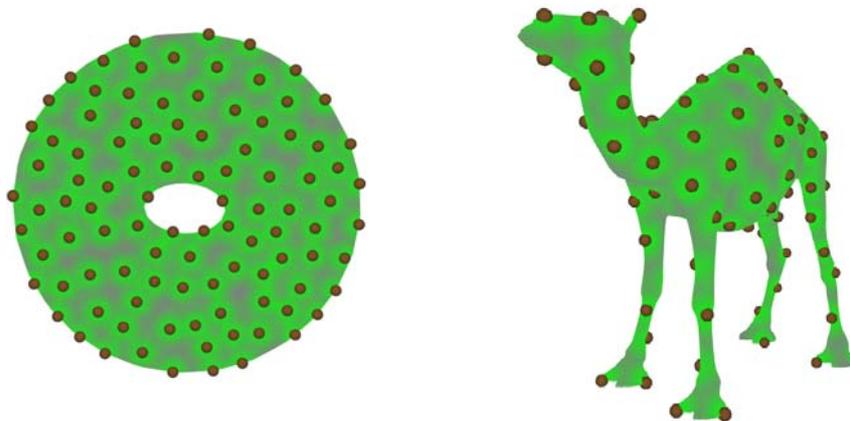
Figure 7.2: One hundred proxy-joints generated by the greedy algorithm for the rest-pose skirt and camel model. Joint influences are also depicted.

At this stage, the rest-pose mesh is ready to be animated with any kind of matrix palette skinning, e.g., linear (Section 2.2.1), spherical (Chapter 3) or dual quaternion skinning (Chapter 4). The only thing that remains to be determined are the actual skinning transformations governing the animation. Their automatic computation is the main contribution of our algorithm and is presented in the next section.

7.2   Fitting of Skinning Transformations

Let us assume that the (proxy-)joints are already given, either designed by animators or generated automatically according to the previous section. The problem now is to find the joint transformations for each keyframe of our input animation. This is done independently for every keyframe; in the following, we therefore describe transformation fitting for one fixed frame. We cannot simply apply the transformation fitting method from [63], because it derives the joint transformations from previously identified quasi-rigid components (while we do not assume the existence of any quasi-rigid components).

For simplicity, we will first consider the case of linear blend skinning. According to our conventions from Section 2.2, the vertex positions in the current frame of the input animation are denoted as $\mathbf{v}_k^{goal}$, $k \in \{1, \ldots, m\}$ (the number of vertices m does not change between frames). The task now is to find the transformations $C_1, \ldots, C_p$, so that the skinning according to Formula (2.6) produces vertices $\mathbf{v}_k'$ as close as possible to $\mathbf{v}_k^{goal}$. All other quantities, i.e., the number of influencing joints n_k, the influencing joint indices $j_{k,1}, \ldots, j_{k,n_k}$ and the weights $w_{k,1}, \ldots, w_{k,n_k}$, are known and fixed. The question is: how general should the class of transformations that we consider be? The simplest option is to consider general affine transformations, because in this case, we can optimize each element in every matrix C_i independently. Let us study this method first.

7.2.1   Affine Transformation Fitting

The problem can be stated as minimization of

$$\sum_{k=1}^{m} \left\| \mathbf{v}_k^{goal} - \mathbf{v}_k' \right\|^2$$

over the joint transformation matrices $C_1, \ldots, C_p$. This is equivalent to the least-squares solution of the linear system

$$\mathbf{v}_k^{goal} = \sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}} \mathbf{v}_k, \ k \in \{1, \ldots, m\} \tag{7.1}$$

with $3m$ equations and $12p$ unknowns (the elements of 3×4 matrices $C_1, \ldots, C_p$). The system from Formula (7.1) can be rewritten as

$$A\mathbf{x} = \mathbf{b}$$

where $\mathbf{x}$ is a $12p$-dimensional unknown vector, A is a $3m \times 12p$ known matrix, and $\mathbf{b}$ is the $3m$-dimensional right-hand side. The matrix A is constructed from vertex weights, rest-pose vertex positions and influencing joints. The vector $\mathbf{b}$ is formed by stacking vertices $\mathbf{v}_1^{goal}, \ldots, \mathbf{v}_m^{goal}$. Note that since each vertex is influenced only by a small number of joints (typically no more than 4), the matrix A will be quite sparse. We obtain the least-squares

solution $\mathbf{x}$ with the LSQR algorithm which exploits the sparsity of the matrix A [104]. The transformation matrices $C_1, \ldots, C_p$ are then extracted from vector $\mathbf{x}$.

Fitting of affine transformations works very well for vertex positions. However, because of realistic shading, we also need to handle vertex normals. For accurate normals, it is not sufficient to simply transform them by the 3×3 submatrices of $C_1, \ldots, C_p$, as observed already in [99]. Instead, it is necessary to incorporate the normals into our fitting process, which can be done in two ways. Either, we can combine the vertex and normal equations together, and find transformations that would be useful for skinning both vertex positions and normals. Alternatively, we can solve two independent systems, one for vertex positions, one for normals, thus computing two sets of transformations. Even though the latter method needs more memory, we found it more advantageous in practice – it also avoids problems with different magnitudes of vertex positions and normals (i.e., vertex magnitudes vary for each particular model, while normals are always unit, which can lead to an unbalanced least-squares fitting). Henceforth, we will therefore consider a separate set of transformations for normals, $C'_1, \ldots, C'_p$.

Vertex normals are transformed in linear blend skinning according to an equation similar to Formula (2.6):

$$\nu'_k = \left(\sum_{i=1}^{n_k} w_{k,i} C'_{j_{k,i}} \right)^{-T} \nu_k \tag{7.2}$$

where $C'_1, \ldots, C'_p$ are the normal transformation matrices. Besides the inverse transposition, the problem is that this equation does not produce unit normals ν'_k, even if the input normals have unit length [99]. This means that we would actually have to minimize

$$\sum_{k=1}^{m} \left\| \nu_k^{goal} - \frac{\nu'_k}{\|\nu'_k\|} \right\|^2 \tag{7.3}$$

which leads to a system of non-linear equations. A non-linear optimization would make our pre-processing times impractical – not to mention the associated numerical issues. Luckily, by constraining our transformations to rigid ones, we are able to obtain a linear least squares problem.

7.2.2 Rigid Transformation Fitting

If we restrict our transformations to rigid ones, and employ a blending method which preserves rigidity, we obtain useful simplifications. Namely, the inverse transposition from Formula (7.2) as well as the normalization in Formula (7.3) disappear, thus linearizing the problem. Moreover, rigid transformations have only 6 degrees of freedom, which means that we need only half the memory required for affine transformations.

On the other hand, fitting rigid transformations is more complex than fitting affine transformations because, in the former case, we must constrain the transformations to be rigid.

The straightforward way to do this would be to require that the 3×3 submatrix of each C_i is orthogonal. Unfortunately, the orthogonality condition is quadratic, and therefore we would again obtain a non-linear optimization problem. In addition to this, linear blend skinning cannot be applied, because we need a blending method that preserves the rigidity of the input transformations (otherwise the transformations of normals would not preserve their unit length).

Fortunately, all of the above-mentioned problems can be elegantly solved by switching to dual quaternion skinning (Chapter 4). Dual quaternions are automatically restricted to rigid transformations, and their blending also naturally preserves rigidity, so the lengths of normal vectors naturally stay unit. Dual quaternion skinning is given by Algorithm 4.2, which can be written shortly in the following way:

$$\hat{\mathbf{v}}'_k = \left(\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}} \right) \hat{\mathbf{v}}_k \left(\overline{\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}}} \right)^{-1} \tag{7.4}$$

where $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_p$ is the dual quaternion representation of matrices $C_1, \ldots, C_p$. One technical difficulty with this equation is that it is only valid if the dual quaternion $\overline{\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}}}$ is invertible. We must thus ensure that our fitting method will produce invertible dual quaternions, i.e., those with a non-zero non-dual part – at least one coefficient of 1, i, j or k must be non-zero. We can enforce this easily by setting the first (real) component of each dual quaternion $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_p$ to one. This does not restrict our set of rigid transformations, because any dual quaternion $\hat{\mathbf{p}}$ represents the same rigid transformation as its real multiple $\alpha \hat{\mathbf{p}}$, for any real number α [94].

Equation (7.4) can be further simplified: if we multiply both sides by $\overline{\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}}}$ from the right and replace $\hat{\mathbf{v}}'_k$ by the desired vertex position $\hat{\mathbf{v}}_k^{goal}$, we obtain:

$$\hat{\mathbf{v}}_k^{goal} \left(\overline{\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}}} \right) - \left(\sum_{i=1}^{n_k} w_{k,i} \hat{\mathbf{q}}_{j_{k,i}} \right) \hat{\mathbf{v}}_k = 0 \tag{7.5}$$

for $k = 1, \ldots, m$. This is a linear system, because multiplication of a dual quaternion by a constant dual quaternion is a linear transformation. Therefore, the above system of equations can be written as

$$A' \mathbf{x}' = \mathbf{b}' \tag{7.6}$$

where $\mathbf{x}'$ is a $7p$-dimensional unknown vector, A' is a $3m \times 7p$ known matrix, and $\mathbf{b}'$ is the $3m$ dimensional right-hand side (which is non-zero due to the substitution of 1 for the real component of dual quaternions $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_p$). Construction of the matrix A' is just a technical matter of rewriting Equation (7.5). The vector $\mathbf{b}'$ is formed from the rest-pose vertices $\mathbf{v}_k$ and the target ones, $\mathbf{v}_k^{goal}$. The matrix A' is sparse and thus Formula (7.6) can be efficiently solved in the least-squares sense using LSQR [104]. This is even faster than affine transformation fitting, because here we have only $7p$ unknowns instead of $12p$. The dual quaternions $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_p$ leading to an optimal fit are then constructed easily: their first component is 1, and the last 7 components are extracted from the vector $\mathbf{x}'$.

The fitting of normals can be done in essentially the same way. The only difference is that, in the case of normals, we ignore the translation component, i.e., set the dual parts of all dual quaternions $\hat{\mathbf{q}}_1, \ldots, \hat{\mathbf{q}}_p$ to zero. This leads to a linear system with only $3p$ unknowns, which we obtain from Equation (7.6) by simply substituting zeros for dual components.

7.2.3 Discussion

Is it more advantageous to use affine or rigid transformation fitting? Intuitively, it would seem that for highly deformable animations, affine transformations should be more appropriate: in particular, they cannot give a worse fit, because rigid transformations are a subset of affine ones. However, our experiments (see Figure 7.5) show that the fitting accuracy of both rigid and affine transformations is actually quite similar. We therefore argue in favor of using rigid transformations, because they need only half as much memory as affine ones, simplify the treatment of normals and also offer faster pre-processing. Theoretically, it would be possible to consider a hybrid system that uses affine transformations for vertex positions and quaternions for vertex normals. However, we find it much more convenient to treat both vertex positions and normals in a unified way. Our final implementation therefore uses dual quaternion-based rigid transformation fitting. Note that James and Twigg [63] arrived at similar conclusions, stating that rigid transformations are more favorable for highly deformable animations than affine ones.

7.3 Adding Fine Details

In some cases, the transformation fitting process described in Section 7.2 (either affine or rigid) has the side effect of smoothing out deformations. This is understandable, because matrix palette skinning simply has insufficient degrees of freedom to reproduce all the fine details of the deformation field. We can increase the accuracy of fitting by adding more proxy-joints, i.e., choosing a higher p. However, some subtle effects, such as delicate wrinkles on the cloth, would require a very high p, thus defeating the purpose of our method. In order to support such effects, we propose an alternative method, based on skinning corrections similar to EigenSkin [74]. This method is most suitable for fine, low-amplitude deformations, and thus presents a perfect complement to our joint transformation fitting, which is most advantageous for low-resolution global shape approximation.

In the rest of this chapter, we will need to work with the animation as a whole, as opposed to the per-frame approach used in Section 7.2. Therefore, κ will denote the number of keyframes. To denote the quantities in keyframe (time) $t = 1, \ldots, \kappa$, we use a superscript ^{t}, for example, $\mathbf{v}_k^t$, $(\mathbf{v}_k^{goal})^t$ and $\hat{\mathbf{q}}_i^t$. Let us assume that we have already computed the joint transformations for each frame according to Section 7.2 (irrespective of whether rigid or affine fitting was used). We denote the final transformation of vertex $\mathbf{v}_k^t$ as T_k^t, e.g., in linear blend skinning, $T_k^t = \sum_{i=1}^{n_k} w_{k,i} C_{j_{k,i}}^t$. The trick of EigenSkin is to transform the approximation error, i.e., the vector $(\mathbf{v}_k^{goal})^t - T_k^t \mathbf{v}_k^t$, to the rest-pose by multiplying it by

$(T_k^t)^{-1}$ from the left. We denote the rest-pose error as $\mathbf{e}_k^t$,

$$\mathbf{e}_k^t = (T_k^t)^{-1}(\mathbf{v}_k^{goal})^t - \mathbf{v}_k^t$$

The vector $\mathbf{e}_k^t$ is the displacement which, if added to $\mathbf{v}_k^t$ before transformation, corrects the skin to match the input exactly (see Figure 7.3).
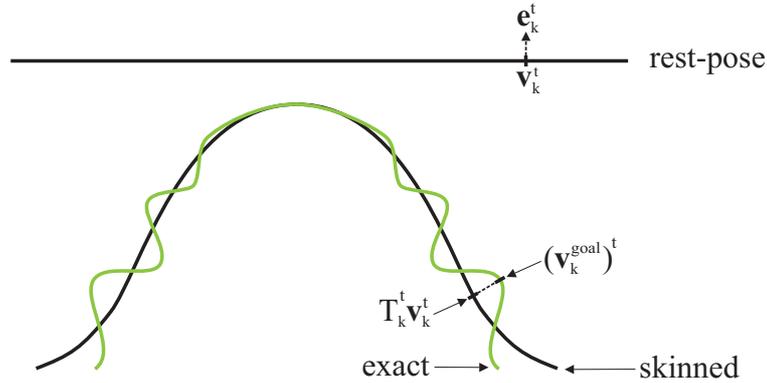
Figure 7.3: Differences between the exact mesh position, $(\mathbf{v}_k^{goal})^t$, and its skinning approximation, $T_k^t \mathbf{v}_k^t$, are mapped to the rest-pose, and denoted as $\mathbf{e}_k^t$.

We stack all vectors $\mathbf{e}_k^t$ in a $3m \times \kappa$ matrix E. Then we apply singular value decomposition to decompose the matrix E to $E = DK$, where D is a $3m \times \kappa$ matrix whose columns are so-called eigen-displacement vectors and K is an $\kappa \times \kappa$ matrix of eigen-displacement coefficients. The reason for this decomposition is that only the first few eigen-displacements are necessary for a good approximation of matrix E (this is a consequence of the high correlation between the columns of matrix E). This means that instead of storing the matrix E, which is as big as the original animation, we store only the first f columns of matrix D and the first f rows of matrix K. An approximation of matrix E, which we denote as E', is then given as a matrix multiplication $E' = D'K'$, where D' and K' have the same size as D and K, but the last $\kappa - f$ columns of matrix D' and last $\kappa - f$ rows of matrix K' are zero. In a practical implementation, the matrix E' is actually never explicitly evaluated. Instead, the elements of E' are computed as needed, and can be efficiently implemented in a vertex shader. When computing the matrix palette skinning, the elements $(\mathbf{e}_k')^t$ of matrix E' are used as corrections to rest-pose vertex positions, i.e., the corrected deformed vertex positions are now computed as

$$\mathbf{v}_k'' = T_k^t(\mathbf{v}_k^t + (\mathbf{e}_k')^t)$$

Typically, even a value of $f = 1$ adds most of the fine details, see Figure 7.4. When using rigid transformation fitting (Section 7.2.2), the normals can be corrected in the same way as vertex positions, giving other correction matrices D_n', K_n' for normals. If allowing non-rigid transformations, we can also use the same scheme, but we must take into account that the normals will be transformed by the inverse transposition of matrix T_k^t in this case.

no corrections   original

1 eigen-disp.   2 eigen-disp.   3 eigen-disp.
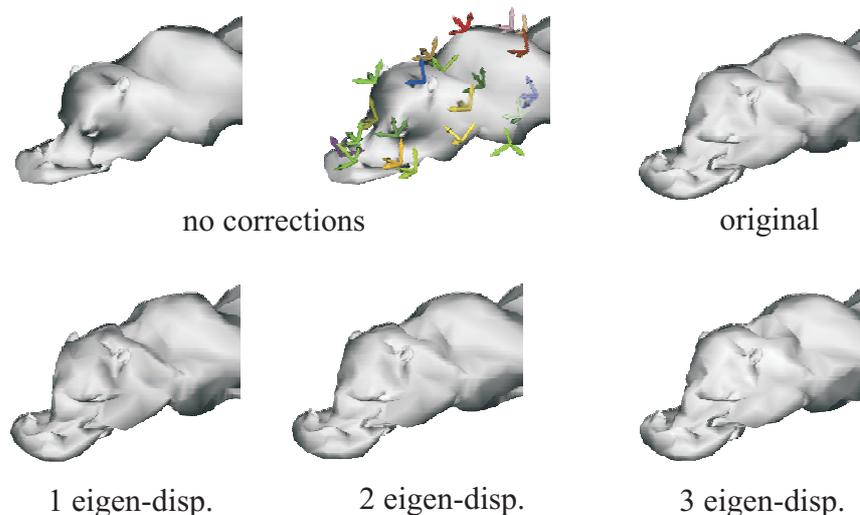
Figure 7.4: EigenSkin corrections [74], adapted to our settings. Matrix palette skinning sometimes smoothes out the deformations, because of lack of samples (transformations). However, fine details can be recovered by adding only few eigen-displacements.

7.4   Experiments and Comparison

In order to obtain comparable results, we use the same error metric for measuring the fitting accuracy as in [63]. This metric is expressed as the percentage of distortion:

$$\%Error = 100 \frac{\|P_{exact} - P_{approx}\|_F}{\|P_{exact} - P_{average}\|_F}$$

where P_{exact} is the $3m \times \kappa$ matrix storing the original animation, P_{approx} is the animation reconstructed using our method and $P_{average}$ is a matrix where every column is the same and is equal to the average of P_{exact} over all columns (keyframes). The symbol $\|\cdot\|_F$ denotes the Frobenius norm of a matrix. Our testing animations (except the elastic hippopotamus and shark) were created in 3D Studio Max using physically based cloth simulation. The elastic hippopotamus and shark are animated in the same software but as FFD soft bodies. For the visual results please see Figures 7.1, 7.7 and 7.9. The results of approximating these animations using 100 proxy-joints are reported in Table 7.1. This relatively high number of proxy-joints is a conservative estimate of how many transformations are really needed. If higher data reduction was necessary, it would be possible to find an optimal number of proxy-joints for each animation (it is not our goal to outperform the previous animation compression algorithms in terms of compression ratio). In practice, 100 rigid transformations per frame are unlikely to present an issue in terms of memory budget or a performance bottleneck (which is much more likely to occur in the vertex or fragment shader, because of the 3D models' sizes).

An important issue is how our method compares to Skinned Mesh Animations (SMAs) [63]. Unfortunately, we could not simply run SMAs on our testing animations, because

Animation	Vertices	Triangles	Keyfr.	Pre-process.	Error	Compress.
Falling Skirt 1	1468	2811	30	2.2 min	0.69	5.9
Falling Skirt 2	4969	9706	60	10.7 min	0.86	17.6
Skirt Walk 1	2963	5747	54	8.2 min	0.84	11.6
Skirt Walk 2	7526	14766	60	19.9 min	1.33	23.1
Curtain	6409	12528	200	48.7 min	0.16	27.6
Elastic Hippopotamus	6442	11542	100	151.8 min	0.11	24.4
Elastic Shark	10070	19301	100	205.7 min	0.04	33.5
Cloth Hippopotamus	6442	11542	100	33.5 min	0.86	24.4
Cloth Camel	20330	28332	100	79.5 min	0.8	50.4

Table 7.1: Results of Skinning Arbitrary Deformations for our testing animations. Conditions: rigid transformation fitting, separate handling of vertex positions and normals, 100 proxy-joints, no corrections.

their implementation is not publicly available. However, we executed our algorithm on the highly deformable animations from the SMA paper (courtesy of Doug L. James). We compare the algorithms in the same setting (both use rigid transformations and the same number of proxy-joints). Note that in this case, the numbers of proxy-joints has not been set by the user (as before), but selected by James and Twigg's algorithm. The results are summarized in Table 7.2. We see that, with the sole exception of the elastic cow animation, our algorithm fits with more than five times higher accuracy. This is true for both uncorrected skinned animations as well as for corrections with 10 eigendisplacements (in brackets). In the case of the elastic cow, our algorithm achieves only a slightly better fit than SMA. We presume this is because the elastic cow is still quite similar to a quasi-articulated object, unlike the remaining cloth models.

Animation	m	$\#tri$	κ	p	SAD pp	SAD % E	SMA pp	SMA % E
Cloth Horse	8431	16843	53	6	3.0 min	7.67 (0.17)	7.7 min	41.7 (0.88)
Flag$_{(32joints)}$	6906	13436	200	32	40.7 min	1.17 (0.44)	9.8 min	21.2 (5.93)
Flag$_{(100joints)}$	6906	13436	200	100	142 min	0.46 (0.17)	16.4 min	2.26 (1.25)
Elastic Cow	2904	5804	204	18	5.3 min	2.48 (1.2)	3.1 min	2.82 (1.54)

m ...	number of vertices	SAD pp ...	pre-processing time of SAD
$\#tri$ ...	number of triangles	SAD % E ...	approximation error of SAD
κ ...	number of keyframes	SMA pp ...	pre-processing time of SMA
p ...	number of proxy-joints	SMA % E ...	approximation error of SMA

Table 7.2: Performance of our algorithm (SAD) executed on the highly deformable animations from the Skinning Mesh Animations (SMA) paper [63]. The numbers in brackets denote the error after correction by 10 eigen-displacements. Compression ratio is not reported because is the same for both SAD and SMA.

Experiments on SMAs with about 100 proxy-joints for extra animations (e.g., the cloth horse animation) were carried out by James [61]. Unfortunately, issues arose with the mean shift implementation (an algorithm used by SMAs), and it was found that it is not robust enough to handle such cases (i.e., with a lot of little clusters). We did not encounter any such robustness issues with our algorithm.

However, it should be noted that while our algorithm is also applicable to quasi-articulated animations, SMAs are more advantageous in this case. They automatically determine the lowest suitable number of proxy-joints, and also have shorter pre-processing times than our algorithm. Our algorithm spends the vast majority of pre-processing time in the optimization process (i.e., the LSQR algorithm [104]). This is because in our algorithm, each transformation in each keyframe undergoes optimization.

An interesting question is how fast the error decreases for increasing numbers of proxy-joints. We computed the error for 1 to 100 proxy-joints, using both rigid and affine transformation fitting (see Figure 7.5). We observe two things: first, the rigid transformations quickly become almost as accurate as the affine ones. Therefore, rigid transformations (Section 7.2.2) are preferred, because they lead to twice as good compression as the affine ones. Second, after about 60 proxy-joints, the approximation error decreases very slowly. This suggests that the mesh corrections described in Section 7.3 are more suitable for adding fine details than increasing the number of proxy-joints.
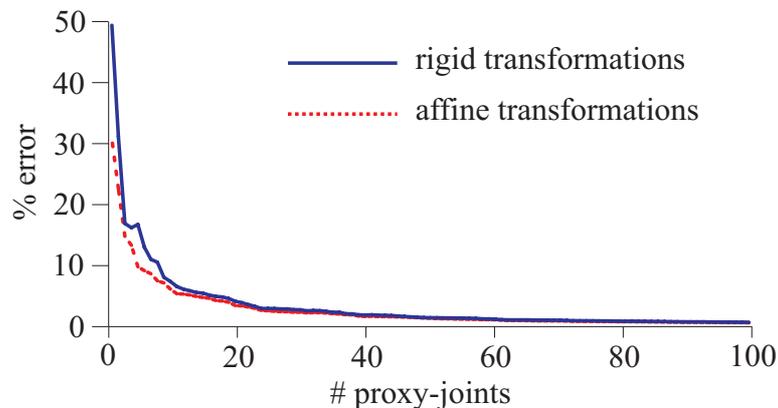
Figure 7.5: Falling Skirt 1 animation: error of fitting for increasing number of proxy-joints. Even for low numbers of proxy-joints, rigid fitting is almost as accurate as affine.

The performance of skin corrections according to Section 7.3 is reported in Figure 7.6. We see that eigen-displacements are indeed a good alternative to increasing the number of proxy-joints, as the fitting error quickly drops to an unobservable level after applying the few first eigen-displacements. This is in accordance with the experiments of Kry et al. [74].

Three frames of the collapsing cloth hippopotamus animation are shown in Figure 7.7. We see that even the rigid transformation fitting without any corrections (top row) approximates the overall shape very well. However, some fine details are smoothed out, e.g., the crease under the hippopotamus' eye – see Figures 7.4 and 7.7. This is improved by
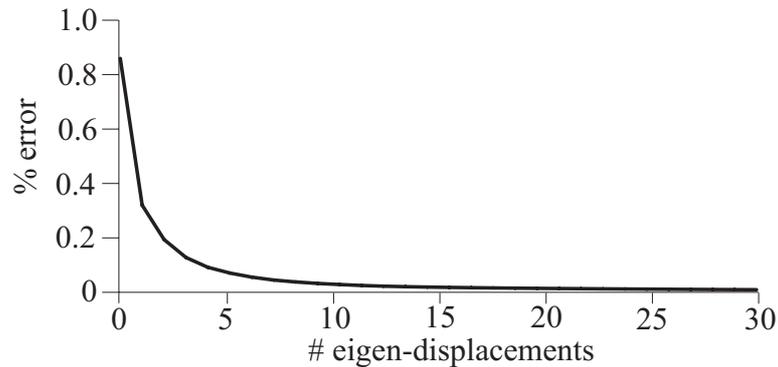
Figure 7.6: The cloth hippopotamus animation (with 100 proxy-joints and rigid transformations) is improved by adding eigen-displacements (Section 7.3). Even one eigen-displacement adds most fine details.

adding 5 corrective eigen-displacements (Figure 7.7 middle row), which makes the reconstruction visually indistinguishable from the original animation (Figure 7.7 bottom). The error of the uncorrected skinning is 0.86% and decreases to 0.07% after correction with 5 eigen-displacements.
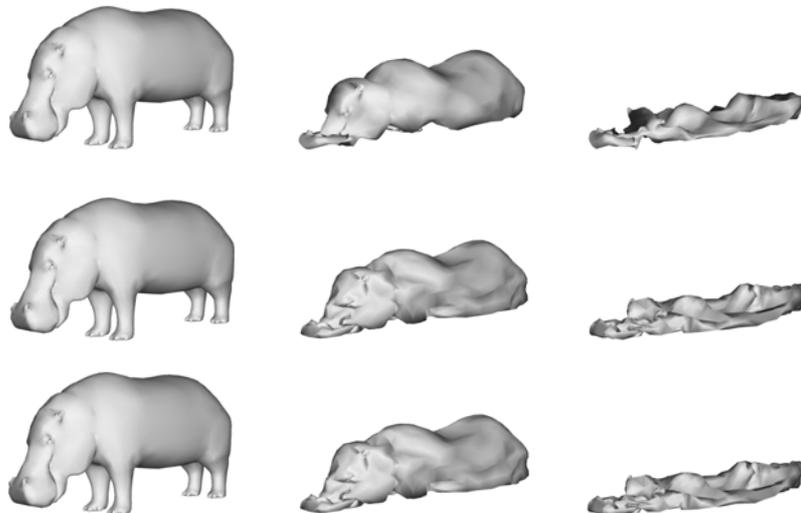
Figure 7.7: The animation of a collapsing cloth hippopotamus. **Top:** Uncorrected skinning with 100 proxy-joints and rigid transformations. **Middle:** Skinning corrected by adding 5 eigen-displacements. **Bottom:** The original animation.

Matrix palette skinning approximations reduce the degrees of freedom of the animation, which is useful in a number of applications, as discussed already by James and Twigg [63]. Our algorithm facilitates efficient hardware accelerated rendering, collision detection and rest-pose editing for non-quasi articulated models. Our performance testing scenario involves one thousand unsimplified elastic shark models (see Figure 7.8). The skinned
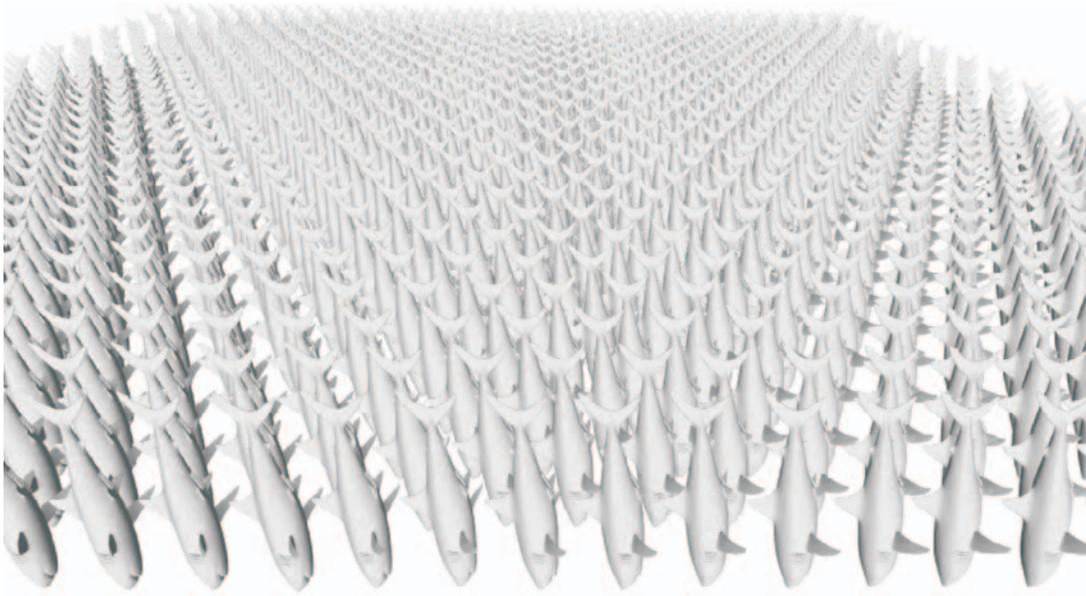
Figure 7.8: One thousand unsimplified shark models, animated using our method on a GeForce 6600 GT at 2.42 FPS.

(a)   (b)

(c)   (d)
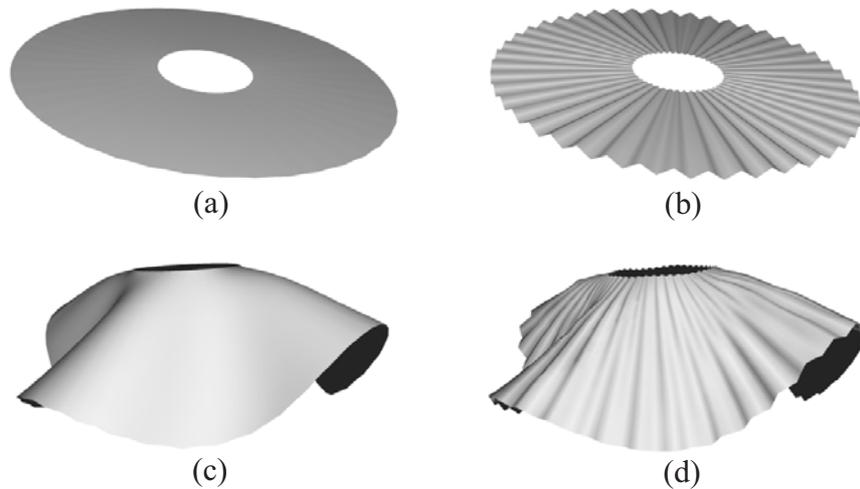
Figure 7.9: Rest-pose editing: the original plain skirt in the rest-pose (a), is changed to a pleated skirt (b). Our method then automatically propagates the pleats over the whole animation. The image (c) shows one frame of the original animation, and (d) the resulting one.

approximations avoid the bottleneck of sending large amounts of data down the graphics pipeline, and therefore allow us to achieve 2.42 FPS instead of 0.64 FPS as in the classic approach (i.e., sending all vertex and normal data for each keyframe). Our implementation also benefits from the rigid transformation fitting, by sending dual quaternions instead of

matrices (which is more efficient because a dual quaternion needs only 8 floats, instead of 12 for a rigid transformation matrix).

The deformable collision detection in [63] is based on a Bounded Deformation Tree [62], constructed for each rigid component. Obviously, this cannot work for our method, because we do not assume the existence of any rigid components. However, the skinned approximation computed by our algorithm can be used for efficient collision detection as described in Chapter 5.

Finally, we demonstrate that our skinning approximations are robust enough to propagate small changes of the rest-pose geometry over the rest of the animation. This is a convenient tool for animation editing and/or simplification. Imagine, for example, that we want to change an original plain skirt into a pleated one. Thanks to our skinning approximation, this can be done simply by editing the rest-pose – see Figure 7.9. Note also that thanks to our simple weighting scheme (Section 7.1), it is also possible to change the mesh connectivity and, for example, perform mesh simplification.

8 Summary and Conclusions

This thesis discusses several techniques to achieve a more realistic animation and simulation of skeletally deformable objects. We focus on one specific class of skeletal deformation algorithms, i.e., those based on blending of rigid transformations. This choice is motivated by both the industrial importance of this model and the lack of its theoretical study. Even though our main goal is to develop algorithms useful in practice, an equal attention has been paid to the related theoretical background, providing more formal justification where necessary. This chapter summarizes our contributions and discusses potential directions of future work.

8.1 Our Contribution in Skinning

In the first part of this thesis (Chapters 3 and 4), we focus on the problem of realistic and efficient skin deformation. We identify the main problem of the standard solution (linear blend skinning) in the fact that it produces non-rigid output transformations. Based on this observation, we describe our first improvement: spherical blend skinning [A.6], which produces rigid transformations and indeed removes most artifacts. As we demonstrate on our experimental 3D objects, spherical blending is only slightly slower than linear blending and can therefore be a practical alternative.

Unfortunately, spherical blending involves a complex algorithm (singular value decomposition) as a subroutine. This presents some implementational difficulties (especially concerning a GPU implementation) and it is probably the reason why spherical blending did not achieve a big popularity among developers. Therefore, we continued by searching method that would have similar properties as spherical blending, but would be easier to implement and friendly to graphics hardware.

To this end, we conducted a theoretical investigation of rigid transformation blending algorithms [A.8]. We formulated mathematical properties that an ideal rigid transformation blending for skinning should fulfill (Section 4.2). Based on our survey, we concluded that no previously described rigid transformation blending algorithm is optimal for skinning. Therefore, we developed new rigid transformation blending algorithms based on dual quaternions. One of these algorithms, dual quaternion linear blending (Section 4.3) has been found to be especially advantageous for skinning. This is because it 1) almost exactly fulfills our set of optimal blending properties and 2) is very efficient and simple to implement on the GPU. We believe that this technique will finally become an alternative to the popular but inaccurate linear blend skinning.

Although our study of rigid transformation blending algorithms originated from the goal to find an optimal blending algorithm for skinning, the resulting methods are general and potentially useful in other contexts as well. The importance of transformation blending in computer graphics has been emphasized by Alexa [3].

As an interesting by-product, this work demonstrates the utility of dual quaternions – a tool that did not obtain too much attention in the computer graphics literature so far (in contrast to regular quaternions). We presume that in future, more applications of dual quaternions in computer graphics can be found.

In Chapter 7, we presented a method to automatically generate skinned approximations of arbitrary deformations. To our knowledge, this is the first attempt to extend the popular matrix palette skinning methods to a more general class of deformations, such as those of cloth and elastic materials. This allows the benefits of skinned animations (e.g., data reduction, efficient hardware accelerated rendering, fast collision detection and rest-pose editing) to be exploited for more general animations than quasi-articulated ones.

8.2   Our Contribution in Collision Detection

Chapters 5 and 6 focus on collision detection algorithms specialized for skeletally deformable objects. Although the specialization on a given deformation model can be criticized for the lack of generality, it has one major benefit: it enables to take advantage of the special properties of the deformation model in question. In our case, this means that collision detection can be based on joint transformations rather than on vertex displacements (which is the classical approach). This enables us to achieve a considerable speed-up over previous techniques, because the number of joints is typically much smaller than the number of vertices. Obviously, no general collision detection technique can offer this advantage: in the case of general deformations, it is inevitable to process all vertices. This presents a bottleneck in real-time simulations, as observed on several experimental scenarios.

Our first collision detection algorithm is designed for linear blend skinning [A.1]. This might seem surprising in the light of our results in advanced skinning methods. However, the problem of efficient sphere refitting (which is the key problem in our collision detection method) is not trivial already in the simple linear blending. Therefore, we have chosen it as the starting point of our investigation. Another justification of collision detection for linear blend skinning is that this skinning method is already approved in the videogames industry.

In order to construct efficient sphere refitting for linear blending, we developed the concept of convex combination of spheres and generalized convex hulls (Section 5.2). We find these tools also theoretically interesting, and we hope that they might be useful in other situations, besides accelerating collision detection. On several benchmarks, we demonstrated that our collision detection algorithm for linear blend skinning significantly outperforms previous generic methods (Section 5.4).

Motivated by these results, we continued by designing a similar collision detection algorithm for spherical blend skinning. One might wonder why we did not opt for the more advantageous dual quaternion skinning. This is because the work on collision detection

for spherical blend skinning has been done before discovering the dual quaternion blending method. Collision detection for dual quaternion skinning would be an interesting work we would like to pursue in the future.

In contrast to the sphere refitting for linear blending, where it is sufficient to design bounds of linear combinations, in the case of spherical blending we have to design spherical (quaternion) bounds. This is accomplished by introducing the concept of spherical cap (Section 6.3). Of course, the possible applications of spherical caps are by no means limited to collision detection. Even though the derivation and justification of spherical bounds is a bit more complicated than in the linear case, the resulting algorithm is almost as efficient as that for linear blending, as has been verified on our experimental scenarios.

From a theoretical viewpoint, we have shown that there exist exact sub-linear collision detection algorithms for both linear and spherical blend skinning.

8.3 Future Work

8.3.1 Collision Detection for Dual Quaternion Blending

A natural continuation of our work on collision detection would be a sub-linear collision detection algorithm for dual quaternion skinning. To this end, it would be necessary to study the geometry of the image space of dual quaternions – a 6-dimensional manifold in 8-dimensional Euclidean space (which can be imagined as the set of tangent planes of a hypersphere [94]). Based on this investigation, efficient bounding volumes would have to be designed in this space. The rest would be equivalent to our previous collision detection algorithms for linear and spherical blend skinning.

8.3.2 Analysis of Dual Quaternion Iterative Blending

In Chapter 4, we omitted a detailed discussion of the dual quaternion iterative blending (DIB) algorithm. This is due to two reasons. First, the DIB algorithm is not as advantageous in skinning as the dual quaternion linear blending (DLB), because of its higher computational complexity (the main purpose of DIB is to show that an algorithm with the desired properties really exists). Second, already the discussion of a simpler spherical case is quite involved (see [22]). Blending on general manifolds would perhaps be a topic for a thesis of itself.

8.3.3 Blending of Non-rigid Transformations

We considered only rigid transformations in our study of transformation blending, because they are the ones most frequently encountered in skinning. However, certain special applications might require usage of general affine transformations (including scale and shear),

for example the animation of muscle bulging. Dual quaternions fall short in representing this kind of transformations and therefore new methods based on more general geometric algebras should be explored. This work can be also very interesting theoretically, both by formulating the desirable properties of affine blending (e.g., finding the analogy of shortest path and constant speed properties when scale and shear is involved) and by designing new algorithms to achieve them. This problem has already been slightly tackled in the recent work of Wareham [133], discussing also some other interesting applications of geometric algebras in computer graphics.

8.3.4   Rendering Performance and Visual Quality

The first part of this thesis focuses mainly on improving the realism of skin deformations. However, recent research records also the opposite trend: improving the run-time performance of character animation [127, 131, 31, 2], sometimes even at the cost of compromising visual quality. Even though linear blend skinning is a very efficient algorithm, its speed might not be sufficient in applications such as rendering of large crowds. Therefore, simplification techniques are sought, using, e.g., less detailed geometries or image based representations. Therefore, another potential area of future research is to improve the performance of rendering many animated objects.

On the other hand, other applications need only few models, but a high level of realism. This might include simulation of muscle bulging and dynamic effects. Even though many dedicated methods exist, it has been shown that this is possible even in the framework of skeletal animation [99, 78]. Perhaps, further convergence of the different approaches to character animation might be achieved in the future.

8.3.5   Adaptive Skinning Arbitrary Deformations

Our method for skinning arbitrary deformations, presented in Chapter 7, distributes the proxy-joints (skinning transformations) uniformly over the rest-pose mesh. This might be sub-optimal for certain types of deformations, e.g., when some parts of the model deform more than others. Therefore, it would be advantageous to consider an adaptive proxy-joint placement scheme, which would assign more proxy-joints to the areas of high deformations. Also, nothing prevents us from changing proxy-joints positions from frame to frame, which could also be exploited for more accurate skinned approximations.

8.4   Conclusion

The presented algorithms might serve as a basis for character animation subsystem in a real-time application – the conducted experiments indicate that our methods can be useful in practice. However, it can be only the real industrial experience that will evaluate the

impact of our proposed techniques. Also, it remains to test our algorithms on a broader set of 3D models, as the data sets available in the industrial environment are of course much more rich than those available for academic purposes.

The problem of skinning, as well as that of character animation, is very young, being developed for only a couple of decades. We are probably yet starting to discover the variety of problems involved and their relationships with other disciplines, such as algebra, geometry and robotics. The industrial importance of skeletal and character animation facilitates and accelerates prospective research – it might happen that the methods presented in this thesis will quickly become outdated. At least, we hope that our techniques will play their role in the unstoppable development.

Bibliography

[1] P. K. Agarwal and M. Sharir. Efficient algorithms for geometric optimization. *ACM Comput. Surv.*, 30(4):412–458, 1998.

[2] J. Ahn, S. Oh, and K. Wohn. Optimized motion simplification for crowd animation: Research articles. *Comput. Animat. Virtual Worlds*, 17(3-4):155–165, 2006.

[3] M. Alexa. Linear combination of transformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 380–387. ACM Press, 2002.

[4] M. Alexa and W. Müller. Representing animations by principal components. *Comput. Graph. Forum*, 19(3):411–418, 2000.

[5] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[6] A. Angelidis. Hexanions: 6D space for twists. Technical report OUCS-2004-20, University of Otago, 2004.

[7] A. Aubel and D. Thalmann. Realistic deformation of human body shapes. *Proc. Computer Animation and Simulation 2000*, pages 125–135, 2000.

[8] A. Aubel and D. Thalmann. Musclebuilder: a modeling tool for human anatomy. *J. Comput. Sci. Technol.*, 19(5):585–595, 2004.

[9] Autodesk. 3D Studio Max. `http://www.autodesk.com`.

[10] Autodesk. Maya. `http://www.autodesk.com`.

[11] C. Babski and D. Thalmann. A seamless shape for hanim compliant bodies. In *VRML '99: Proceedings of the fourth symposium on Virtual reality modeling language*, pages 21–28, New York, NY, USA, 1999. ACM Press.

[12] C. Belta and V. Kumar. An SVD-based projection method for interpolation on SE(3). *IEEE Transactions on Robotics and Automation*, 18(3):334–345, 2002.

[13] C. Bloom, J. Blow, and C. Muratori. Errors and omissions in Marc Alexa's Linear combination of transformations. `http://www.cbloom.com/3d/techdocs/lcot_errors.pdf`, 2004.

[14] J. Bloomenthal. Medial-based vertex deformation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 147–151. ACM Press, 2002.

[15] J. Bloomenthal and J. Rokne. Homogeneous coordinates. *Vis. Comput.*, 11(1):15–26, 1994.

[16] O. Bottema and B. Roth. *Theoretical kinematics*. North-Holland Publishing Company, Amsterdam, New York, Oxford, 1979.

[17] G. Bradshaw and C. O'Sullivan. Sphere-tree construction using dynamic medial axis approximation. In *SCA '02: Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 33–40. ACM Press, 2002.

[18] G. Bradshaw and C. O'Sullivan. Adaptive medial-axis approximation for sphere-tree construction. *ACM Trans. Graph.*, 23(1):1–26, 2004.

[19] H. M. Briceno, P. V. Sander, L. McMillan, S. Gortler, and H. Hoppe. Geometry videos: a new representation for 3D animations. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 136–146, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association.

[20] J. Brown, S. Sorkin, C. Bruyns, J.-C. Latombe, K. Montgomery, and M. Stephanides. Real-time simulation of deformable objects: Tools and application. In *Computer Animation 2001*, pages 228–236, Nov. 2001.

[21] J. Buchanan. Invited talk at ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games, 2005.

[22] S. R. Buss and J. P. Fillmore. Spherical averages and applications to spherical splines and interpolation. *ACM Trans. Graph.*, 20(2):95–126, 2001.

[23] S. Capell, S. Green, B. Curless, T. Duchamp, and Z. Popovic. Interactive skeleton-driven dynamic deformations. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 586–593, New York, NY, USA, 2002. ACM Press.

[24] B. Char, K. Geddes, and G. Gonnet. The Maple symbolic computation system. *j-SIGSAM*, 17(3–4):31–42, Aug./Nov. 1983.

[25] Y.-J. Choi, Y. J. Kim, and M.-H. Kim. Self-CD: Interactive self-collision detection for deformable body simulation using gpus. In *AsiaSim*, pages 187–196, 2004.

[26] W. Clifford. *Mathematical Papers*. London, Macmillan, 1882.

[27] G. Collins and A. Hilton. A rigid transform basis for animation compression and level of detail. In *Vision, Video, and Graphics*, pages 1–7, 2005.

[28] F. Cordier and N. Magnenat-Thalmann. A data-driven approach for real-time clothes simulation. *Computer Graphics Forum*, 24(2):173–183, 2005.

[29] E. Dam, M. Koch, and M. Lillholm. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, University of Copenhagen, 1998.

[30] K. Daniilidis. Hand-eye calibration using dual quaternions. *International Journal of Robotics Research*, 18:286–298, 1999.

[31] S. Dobbyn, J. Hamill, K. O'Conor, and C. O'Sullivan. Geopostors: a real-time geometry / impostor crowd rendering system. In *SI3D '05*, pages 95–102. ACM Press, 2005.

[32] D. Eberly. *3D game engine design: a practical approach to real-time computer graphics.* Morgan Kaufmann Publishers Inc., 2001.

[33] C. Ericson. *Real-Time Collision Detection.* Morgan Kaufmann Publishers Inc., 2004.

[34] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer graphics: principles and practice (2nd ed.).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

[35] D. Fontijne and L. Dorst. Modeling 3D euclidean geometry. *IEEE Comput. Graph. Appl.*, 23(2):68–78, 2003.

[36] B. Gaertner. Fast and robust smallest enclosing balls. In *ESA '99: Proceedings of the 7th Annual European Symposium on Algorithms*, pages 325–338. Springer-Verlag, 1999.

[37] F. Ganovelli, J. Dingliana, and C. O'Sullivan. BucketTree: Improving collision detection between deformable objects. In *Proceedings of the 16th Spring Conference on Computer Graphics*, pages 156–163. Comenius University, Bratislava, 2000.

[38] T. Gonzales. Clustering to minimize the maximum intercluster distance. *Theor. Comput. Sci.*, 38(22):293–306, 1985.

[39] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180, 1996.

[40] N. K. Govindaraju, I. Kabul, M. Lin, and D. Manocha. Fast continuous collision detection among deformable models using graphics processors. In *Eurographics Virtual Environments*, pages 19–26, 2006.

[41] N. K. Govindaraju, D. Knott, N. Jain, I. Kabul, R. Tamstorf, R. Gayle, M. C. Lin, and D. Manocha. Interactive collision detection between deformable models using chromatic decomposition. *ACM Trans. Graph.*, 24(3):991–999, 2005.

[42] N. K. Govindaraju, M. C. Lin, and D. Manocha. Fast and reliable collision culling using graphics hardware. In *VRST '04: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 2–9, New York, NY, USA, 2004. ACM Press.

[43] N. K. Govindaraju, M. C. Lin, and D. Manocha. Quick-cullide: Fast inter- and intra-object collision culling using graphics hardware. In *VR '05: Proceedings of the 2005 IEEE Conference 2005 on Virtual Reality*, pages 59–66, 319, Washington, DC, USA, 2005. IEEE Computer Society.

[44] N. K. Govindaraju, S. Redon, M. C. Lin, and D. Manocha. Cullide: interactive collision detection between complex models in large environments using graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 25–32, Aire-la-Ville, Switzerland, 2003. Eurographics Association.

[45] V. M. Govindu. Lie-algebraic averaging for globally consistent motion estimation. In *CVPR (1)*, pages 684–691, 2004.

[46] F. S. Grassia. Practical parameterization of rotations using the exponential map. *Journal of Graphics Tools*, 3(3):29–48, 1998.

[47] X. Gu, S. J. Gortler, and H. Hoppe. Geometry images. In *SIGGRAPH '02: Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 355–361, New York, NY, USA, 2002. ACM Press.

[48] Z. Guo and K. C. Wong. Skinning with deformable chunks. *Computer Graphics Forum*, 24(3):373–381, 2005.

[49] I. Guskov and A. Khodakovsky. Wavelet compression of parametrically coherent mesh sequences. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 183–192, New York, NY, USA, 2004. ACM Press.

[50] W. R. Hamilton. On quaternions. In *Proceedings of the Royal Irish Academy*, 1844.

[51] A. J. Hanson. *Visualizing Quaternions*. Morgan Kaufmann Publishers Inc., 2006.

[52] J. C. Hart, G. K. Francis, and L. H. Kauffman. Visualizing quaternion rotation. *ACM Trans. Graph.*, 13(3):256–276, 1994.

[53] A. Hawkins and C. Grimm. Keyframing using linear interpolation of matrices. *Journal of Graphics Tools*, 2006.

[54] T. He. Fast collision detection using QuOSPO trees. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 55–62. ACM Press, 1999.

[55] B. Heidelberger, M. Teschner, and M. Gross. Detection of collisions and self-collisions using image-space techniques. In *Proceedings of Computer Graphics, Visualization and Computer Vision WSCG'04*, pages 145–152, 2004.

[56] J. Hejl. Hardware skinning with quaternions. *Game Programming Gems 4*, Charles River Media, 487–495, 2004.

[57] D. Hildenbrand, D. Fontijne, C. Perwass, and L. Dorst. Geometric algebra and its application to computer graphics. EG 2004 tutorial #3, 2004.

[58] V. Houska. Animation using GPU. Master's thesis, Charles University, 2005.

[59] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Trans. Graph.*, 15(3):179–210, 1996.

[60] D.-E. Hyun, S.-H. Yoon, J.-W. Chang, J.-K. Seong, M.-S. Kim, and B. Jüttler. Sweep-based human deformation. *The Visual Computer*, 21(8-10):542–550, 2005.

[61] D. L. James. Personal communication, 2006.

[62] D. L. James and D. K. Pai. BD-Tree: output-sensitive collision detection for reduced deformable models. *ACM Trans. Graph.*, 23(3):393–398, 2004.

[63] D. L. James and C. D. Twigg. Skinning mesh animations. *ACM Trans. Graph.*, 24(3):399–407, 2005.

[64] S. Jianhua, N. Magnenat-Thalmann, and D. Thalmann. Human skin deformation from cross sections. *Proc. Computer Graphics International '94*, 1994.

[65] P. Jiménez, F. Thomas, and C. Torras. 3D collision detection: a survey. *Computers & Graphics*, 25(2):269–285, 2001.

[66] M. P. Johnson. *Exploiting Quaternions to Support Expressive Interactive Character Motion*. PhD thesis, MIT, 2003.

[67] P. Kalra, N. Magnenat-Thalmann, L. Moccozet, G. Sannier, A. Aubel, and D. Thalmann. Real-time animation of realistic virtual humans. *IEEE Comput. Graph. Appl.*, 18(5):42–56, 1998.

[68] A. Karger and J. Novak. *Space Kinematics and Lie Groups*. Gordon and Breach Science Publishers, New York, USA, 1985.

[69] Z. Karni and C. Gotsman. Compression of soft-body animation sequences. *Computers & Graphics*, 28(1):25–34, 2004.

[70] I. Kenneth E. Hoff, A. Zaferakis, M. Lin, and D. Manocha. Fast and simple 2d geometric proximity queries using graphics hardware. In *SI3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 145–148, New York, NY, USA, 2001. ACM Press.

[71] S. Kircher and M. Garland. Progressive multiresolution meshes for deforming surfaces. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 191–200, New York, NY, USA, 2005. ACM Press.

[72] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[73] T. Klug and M. Alexa. Bounding volumes for linearly interpolated shapes. In *Computer Graphics International*, pages 134–139, 2004.

[74] P. G. Kry, D. L. James, and D. K. Pai. Eigenskin: real time large deformation character skinning in hardware. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 153–159. ACM Press, 2002.

[75] T. Kurihara and N. Miyata. Modeling deformable human hands from medical images. In *SCA '04: Proceedings of the 2004 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 355–363, New York, NY, USA, 2004. ACM Press.

[76] J. Lander. Skin them bones: Game programming for the web generation. *Game Developer Magazine*, pages 11–16, May 1998.

[77] J. Lander. Over my dead, polygonal body. *Game Developer Magazine*, pages 17–22, October 1999.

[78] C. Larboulette, M.-P. Cani, and B. Arnaldi. Dynamic skinning: adding real-time dynamic effects to an existing character animation. In *SCCG '05: Proceedings of the 21st spring conference on Computer graphics*, pages 87–93, New York, NY, USA, 2005. ACM Press.

[79] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast proximity queries with swept sphere volumes. Technical report TR99-018, University of N. Carolina, Chapel Hill, 1999.

[80] T. Larsson and T. Akenine-Moller. Collision detection for continuously deforming bodies. In *Eurographics 2001, Short Presentations*, pages 325–333. Eurographics Association, 9 2001.

[81] T. Larsson and T. Akenine-Moller. Efficient collision detection for models deformed by morphing. *The Visual Computer*, 19(2–3):164–174, 2003.

[82] T. Larsson and T. Akenine-Moller. A dynamic bounding volume hierarchy for generalized collision detection. In *Proceedings of the 2nd Workshop on Virtual Reality Interactions and Physical Simulations*, pages 91–100, 2005.

[83] T. Larsson and T. Akenine-Möller. A dynamic bounding volume hierarchy for generalized collision detection. *Computers & Graphics*, 30(3):451–460, June 2006.

[84] J. E. Lengyel. Compression of time-dependent geometry. In *SI3D '99: Proceedings of the 1999 symposium on Interactive 3D graphics*, pages 89–95, New York, NY, USA, 1999. ACM Press.

[85] J. P. Lewis, M. Cordner, and N. Fong. Pose space deformation: a unified approach to shape interpolation and skeleton-driven deformation. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*, pages 165–172. ACM Press/Addison-Wesley Publishing Co., 2000.

[86] J. Li and P. Hao. Smooth interpolation on homogeneous matrix groups for computer animation. *Journal of Zhejiang University*, 7(7):1168–1177, 2006.

[87] E. Lindholm, M. J. Kligard, and H. Moreton. A user-programmable vertex engine. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 149–158, New York, NY, USA, 2001. ACM Press.

[88] N. Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Comput. Graph. Forum*, 13(3):155–166, 1994.

[89] N. Magnenat-Thalmann, F. Cordier, H. Seo, and G. Papagianakis. Modeling of bodies and clothes for virtual environments. In *CW '04: Proceedings of the 2004 International Conference on Cyberworlds (CW'04)*, pages 201–208. IEEE Computer Society, 2004.

[90] N. Magnenat-Thalmann, R. Laperrière, and D. Thalmann. Joint-dependent local deformations for hand animation and object grasping. In *Proceedings on Graphics interface '88*, pages 26–33. Canadian Information Processing Society, 1988.

[91] K. Mamou, T. Zaharia, and F. Preteux. A skinning approach for dynamic 3D mesh compression: Research articles. *Comput. Animat. Virtual Worlds*, 17(3-4):337–346, 2006.

[92] A. Manganas, M. Tsiknakis, E. Leisch, M. Ponder, T. Molet, B. Herbelin, N. Magnenat-Thalmann, and D. Thalmann. Just in time health emergency interventions: An innovative approach to training the citizen for emergency situations using virtual reality techniques and advanced it tools. *The Journal on Information Technology in Healthcare*, pages 27–37, 2005.

[93] J. Matousek. *Lectures on Discrete Geometry*. Springer, April 2002.

[94] J. M. McCarthy. *Introduction to theoretical kinematics*. MIT Press, Cambridge, MA, USA, 1990.

[95] C. Mendoza and C. O'Sullivan. An interruptible algorithm for collision detection between deformable objects. In *Proceedings of the 2nd Workshop on Virtual Reality Interactions and Physical Simulations*, pages 73–80, 2005.

[96] B. Merry, P. Marais, and J. Gain. Animation space: A truly linear framework for character animation. *ACM Trans. Graph.*, 25(4):1400–1423, 2006.

[97] J. Mezger, S. Kimmerle, and O. Etzmuß. Hierarchical Techniques in Collision Detection for Cloth Animation. *Journal of WSCG*, 11(2):322–329, 2003.

[98] M. Moakher. Means and averaging in the group of rotations. *SIAM Journal on Matrix Analysis and Applications*, 24(1):1–16, 2002.

[99] A. Mohr and M. Gleicher. Building efficient, accurate character skins from examples. *ACM Trans. Graph.*, 22(3):562–568, 2003.

[100] A. Mohr, L. Tokheim, and M. Gleicher. Direct manipulation of interactive character skins. In *Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 27–30. ACM Press, 2003.

[101] R. M. Murray, S. S. Sastry, and L. Zexiang. *A Mathematical Introduction to Robotic Manipulation*. CRC Press, Inc., Boca Raton, FL, USA, 413–414, 1994.

[102] S. Oh, H. Kim, N. Magnenat-Thalmann, and K. Wohn. Generating unified model for dressed virtual humans. *The Visual Computer*, 21(8-10):522–531, 2005.

[103] R. Ott, D. Thalmann, and F. Vexo. Organic shape modelling. *Computer-Aided Design and Applications*, 3(1–4):79–88, 2006.

[104] C. C. Paige and M. A. Saunders. Algorithm 583: LSQR: Sparse linear equations and least squares problems. *ACM Trans. Math. Softw.*, 8(2):195–209, 1982.

[105] S. I. Park, H. J. Shin, and S. Y. Shin. On-line locomotion generation based on motion blending. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 105–111. ACM Press, 2002.

[106] M. Pratscher, P. Coleman, J. Laszlo, and K. Singh. Outside-in anatomy based character rigging. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 329–338, New York, NY, USA, 2005. ACM Press.

[107] X. Provot. Collision and self-collision handling in cloth model dedicated to design garments. In *Proceedings of the Eurographics Workshop on Computer Animation and Simulation (CAS 1997)*, pages 177–189. Springer-Verlag, 1997.

[108] S. Quinlan. Efficient distance computation between non-convex objects. In *ICRA*, pages 3324–3329, 1994.

[109] S. Redon, A. Kheddar, and S. Coquillart. An algebraic solution to the problem of collision detection for rigid polyhedral objects. In *Proceedings of IEEE International Conference on Robotics and Automation*, pages 3733–3738, 2000.

[110] S. Redon, A. Kheddar, and S. Coquillart. Contact: Arbitrary in-between motions for collision detection. In *Proceedings of IEEE International Workshop on Robot-Human Communication*, pages 106–111, 2001.

[111] S. Redon, A. Kheddar, and S. Coquillart. Fast continuous collision detection between rigid bodies. *Comput. Graph. Forum*, 21(3):279–288, 2002.

[112] S. Redon, Y. J. Kim, M. C. Lin, and D. Manocha. Fast continuous collision detection for articulated models. In *Proceedings of ACM Symposium on Solid Modeling and Applications*, pages 126–137, 2004.

[113] S. Redon, Y. J. Kim, M. C. Lin, D. Manocha, and J. Templeman. Interactive and continuous collision detection for avatars in virtual environments. In *VR '04: Proceedings of the IEEE Virtual Reality 2004 (VR'04)*, pages 117–124. IEEE Computer Society, 2004.

[114] T. Rhee, J. Lewis, and U. Neumann. Real-time weighted pose-space deformation on the GPU. *Computer Graphics Forum*, 25(3):439–448, 2006.

[115] M. Sattler, R. Sarlette, and R. Klein. Simple and efficient compression of animation sequences. In *SCA '05: Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 209–217, New York, NY, USA, 2005. ACM Press.

[116] T. W. Sederberg and S. R. Parry. Free-form deformation of solid geometric models. In *SIGGRAPH '86: Proceedings of the 13th annual conference on Computer graphics and interactive techniques*, pages 151–160, New York, NY, USA, 1986. ACM Press.

[117] K. Shoemake. Animating rotation with quaternion curves. In *Proceedings of the 12th annual conference on Computer graphics and interactive techniques*, pages 245–254. ACM Press, 1985.

[118] D. Shreiner, M. Woo, J. Neider, and T. Davis. *OpenGL(R) Programming Guide : The Official Guide to Learning OpenGL(R), Version 2 (5th Edition)*. Addison-Wesley Professional, August 2005.

[119] K. Singh and E. Kokkevis. Skinning characters using surface oriented free-form deformations. In *Graphics Interface*, pages 35–42, May 2000.

[120] P.-P. J. Sloan, C. F. Rose, III, and M. F. Cohen. Shape by example. In *Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 135–143. ACM Press, 2001.

[121] J. Snyder. Interval analysis for computer graphics. In *Computer Graphics*, volume 26, pages 121–130, 1992.

[122] P. Steed. *Animating Real-Time Game Characters with CDROM*. Charles River Media, 2002.

[123] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized spatial hashing for collision detection of deformable objects. In *Proc. Vision, Modeling, Visualization VMV'03*, pages 47–54, Munich, Germany, 2003.

[124] M. Teschner, S. Kimmerle, G. Zachmann, B. Heidelberger, L. Raghupathi, A. Fuhrmann, M.-P. Cani, F. Faure, N. Magnetat-Thalmann, and W. Strasser. Collision detection for deformable objects. In *Proc. Eurographics, State-of-the-Art Report*, pages 119–135, Grenoble, France, 2004. Eurographics Association.

[125] D. Thalmann, J. Shen, and E. Chauvineau. Fast realistic human body deformations for animation and VR applications. In *CGI '96: Proceedings of the 1996 Conference on Computer Graphics International*, pages 166–176, Washington, DC, USA, 1996. IEEE Computer Society.

[126] Turbo Squid. 3D models repository. `http://www.turbosquid.com`.

[127] B. Ulicny and D. Thalmann. Crowd simulation for interactive virtual environments and vr training systems. In *Proceedings of the Eurographic workshop on Computer animation and simulation*, pages 163–170, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[128] G. van den Bergen. Efficient collision detection of complex deformable models using AABB trees. *Journal of Graphics Tools: JGT*, 2(4):1–14, 1997.

[129] T. I. Vassilev, B. Spanlang, and Y. Chrysanthou. Fast cloth animation on walking avatars. *Comput. Graph. Forum*, 20(3):260–267, 2001.

[130] P. Volino and N. Magnenat-Thalmann. Efficient self-collision detection on smoothly discretized surface animations using geometrical shape regularity. *Comput. Graph. Forum*, 13(3):155–166, 1994.

[131] M. Wand and W. Straßer. Multi-resolution rendering of complex animated scenes. *Computer Graphics Forum*, 21(3):483–491, 2002.

[132] X. C. Wang and C. Phillips. Multi-weight enveloping: least-squares approximation techniques for skin animation. In *Proceedings of the 2002 ACM SIGGRAPH/Eurographics symposium on Computer animation*, pages 129–138. ACM Press, 2002.

[133] R. Wareham, J. Cameron, and J. Lasenby. Applications of conformal geometric algebra in computer vision and graphics. *Lecture Notes in Computer Science*, 3519:329–349, 2005.

[134] J. Weber. Run-time skin deformation. In *Proceedings of Game Developers Conference*, 2000.

[135] X. Yang and J. J. Zhang. Stretch it - realistic smooth skinning. In *Computer Graphics, Imaging and Visualization*, pages 323–328, Los Alamitos, CA, USA, 2006. IEEE Computer Society.

[136] G. Zachmann. Minimal hierarchical collision detection. In *VRST '02: Proceedings of the ACM symposium on Virtual reality software and technology*, pages 121–128, New York, NY, USA, 2002. ACM Press.

[137] L. Zuo, J.-T. Li, and Z.-Q. Wang. Anatomical human musculature modeling for real-time deformation. In *Proceedings of Computer Graphics, Visualization and Computer Vision WSCG*, 2003.

Publications of the Author

Refereed Journal Papers

[A.1] L. Kavan, J. Žára. Fast Collision Detection for Skeletally Deformable Models. *Computer Graphics Forum*, 24(3):363–372, 2005.

Reviewed Conference Papers

[A.2] L. Kavan, S. Collins, C. O'Sullivan, J. Žára. Skinning with Dual Quaternions. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007 (to appear).

[A.3] L. Kavan, R. McDonnell, S. Dobbyn, J. Žára, C. O'Sullivan. Skinning Arbitrary Deformations. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, 2007 (to appear).

[A.4] L. Kavan, J. Žára, C. O'Sullivan. Efficient Collision Detection for Spherical Blend Skinning. *GRAPHITE, Computer graphics and interactive techniques in Australasia and South East Asia*, ACM Press, pages 147–156, 2006.

[A.5] S. Dobbyn, R. McDonnell, L. Kavan, S. Collins, C. O'Sullivan. Clothing the Masses: Real-Time Clothed Crowds With Variation. *Eurographics 2006 Short Papers Proceedings*, pages 103–106, 2006.

[A.6] L. Kavan, J. Žára. Spherical Blend Skinning: A Real-time Deformation of Articulated Models. *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games*, pages 9–16, 2005.

[A.7] L. Kavan, J. Žára. Real-time Skin Deformation with Bones Blending. *Winter School on Computer Graphics 2003 Short papers*, pages 69–74, 2003.

Other Publications

[A.8] L. Kavan, S. Collins, C. O'Sullivan, J. Žára. Dual Quaternions for Rigid Transformation Blending. Research Report TCD-CS-2006-46, Computer Science Department, Trinity College Dublin, August 2006.

[A.9] L. Kavan, R. McDonnell, S. Dobbyn, J. Žára, C. O'Sullivan. Skinning Arbitrary Deformations. *ACM SIGGRAPH/Eurographics Symposium on Computer Animation, Posters*, 2006.

[A.10] L. Kavan. Simulation of Fencing in Virtual Reality, Master's thesis, Charles University, Prague, September 2003.

A Detailed Proofs

Two statements in this thesis require a rather lengthy verification. In order to maintain the continuity of the text in the previous chapters, we present the proofs in this appendix. Some tedious derivations in these proofs are carried out using Maple (a software package for automatic manipulation with symbolic expressions [24]).

A.1   Difference between DLB and ScLERP

As stated in Section 4.3, the problem of comparing DLB and ScLERP requires to express the dual angle $\hat{\beta}_t$ from the formula

$$\cos\frac{\hat{\beta}_t}{2} = \frac{1 - t + t\cos(\frac{\hat{\alpha}}{2})}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|}$$

and then compare it with $\hat{\alpha}t$ for $t \in [0, 1]$. Before switching to Maple, we start with some manual simplifications. First, let us recall the formulas for dual number sine and cosine:

$$\cos\left(\frac{\hat{\alpha}}{2}\right) = \cos\left(\frac{\alpha_0}{2}\right) - \epsilon\frac{\alpha_\epsilon}{2}\sin\left(\frac{\alpha_0}{2}\right)$$

$$\sin\left(\frac{\hat{\alpha}}{2}\right) = \sin\left(\frac{\alpha_0}{2}\right) + \epsilon\frac{\alpha_\epsilon}{2}\cos\left(\frac{\alpha_0}{2}\right)$$

see [30]. For brevity, we will use shortcut C for $\cos(\frac{\alpha_0}{2})$ and S for $\sin(\frac{\alpha_0}{2})$. Using this shorthand, we derive:

$$\begin{aligned}
1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}} &= 1 - t + t\cos\left(\frac{\hat{\alpha}}{2}\right) + \hat{\mathbf{n}}t\sin\left(\frac{\hat{\alpha}}{2}\right) = \\
&= 1 - t + tC - \epsilon t\frac{\alpha_\epsilon}{2}S + (\mathbf{n}_0 + \epsilon\mathbf{n}_\epsilon)t\left(S + \epsilon\frac{\alpha_\epsilon}{2}C\right) = \\
&= \underbrace{1 - t + tC + \mathbf{n}_0 tS}_{\mathbf{r}_0} + \epsilon\underbrace{t\left(-\frac{\alpha_\epsilon}{2}S + \mathbf{n}_\epsilon S + \frac{\alpha_\epsilon}{2}\mathbf{n}_0 C\right)}_{\mathbf{r}_\epsilon}
\end{aligned}$$

The newly introduced vectors $\mathbf{r}_0$ and $\mathbf{r}_\epsilon$ satisfy

$$\begin{aligned}
\|\hat{\mathbf{r}}_0\| &= \sqrt{(1 - t + tC)^2 + t^2 S^2} \\
\langle\hat{\mathbf{r}}_0, \hat{\mathbf{r}}_\epsilon\rangle &= \left\langle 1 - t + tC + \mathbf{n}_0 tS, -t\frac{\alpha_\epsilon}{2}S + t\mathbf{n}_\epsilon S + t\frac{\alpha_\epsilon}{2}\mathbf{n}_0 C\right\rangle \\
&= (t - 1 - tC)t\frac{\alpha_\epsilon}{2}S + \frac{\alpha_\epsilon}{2}t^2 CS = (t - 1)t\frac{\alpha_\epsilon}{2}S
\end{aligned}$$

because $\hat{\mathbf{n}} = \mathbf{n}_0 + \epsilon\mathbf{n}_\epsilon$ is unit dual quaternion (with zero scalar part). Therefore, the denominator of our equation can be written as

$$\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\| = \|\hat{\mathbf{r}}_0 + \epsilon\hat{\mathbf{r}}_\epsilon\| = \|\hat{\mathbf{r}}_0\| + \epsilon\frac{\langle\hat{\mathbf{r}}_0, \hat{\mathbf{r}}_\epsilon\rangle}{\|\hat{\mathbf{r}}_0\|}$$

and its inverse

$$\frac{1}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|} = \frac{1}{\|\hat{\mathbf{r}}_0\|} - \epsilon\frac{\langle\hat{\mathbf{r}}_0, \hat{\mathbf{r}}_\epsilon\rangle}{\|\hat{\mathbf{r}}_0\|^3}$$

which gives

$$\frac{1 - t + t\cos(\frac{\hat{\alpha}}{2})}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|} = \frac{1 - t + tC - \epsilon t\frac{\alpha_\epsilon}{2}S}{\|1 - t + t\hat{\mathbf{p}}^*\hat{\mathbf{q}}\|} = \frac{1 - t + tC}{\|\hat{\mathbf{r}}_0\|} - \epsilon\left(\frac{t\alpha_\epsilon S}{2\|\hat{\mathbf{r}}_0\|} + \frac{(1 - t + tC)\langle\hat{\mathbf{r}}_0, \hat{\mathbf{r}}_\epsilon\rangle}{\|\hat{\mathbf{r}}_0\|^3}\right)$$

We denote the function above as $f(t)$. Now, we feel inevitable to employ Maple in order to compute $\hat{\beta}_t$ by taking the arccos of $f(t)$. In the following listing, the norm $\|\hat{\mathbf{r}}_0\|$ is denoted as `r0`, non-dual and dual parts of $f(t)$ as `f0` and `fe`. The non-dual component of $\hat{\beta}_t$ is called `ang` and the dual one `pitch`, emphasizing their geometric interpretation. In the first part of the listing, we actually re-compute the result from Section 3.2, showing that the maximal angular difference between QLB and SLERP is 0.143 radians, i.e., 8.15 degrees. In the second part (the dual quaternion-specific one), we derive the difference between the translational parts of DLB and ScLERP, which turns out to be a linear function of α_ϵ (the input translation). Specifically, the difference between translation of DLB and ScLERP shows to be always strictly less than $0.151\alpha_\epsilon$.

```
> r0 := sqrt((1-t+t*cos(alpha_0/2))^2 + t*t*sin(alpha_0/2)^2 ):
> f0 := (t*cos(alpha_0/2) + 1-t)/r0:
> fe := - t*alpha_e/2*sin(alpha_0/2)/r0 - (1-t +
> t*cos(alpha_0/2))*(t-1)*t*alpha_e/2*sin(alpha_0/2) / r0^3:
> ang := 2*arccos(f0);
```

$$ang := 2\arccos\left(\frac{1 - t + t\cos(\frac{1}{2}\,alpha\_0)}{\sqrt{(1 - t + t\cos(\frac{1}{2}\,alpha\_0))^2 + t^2\sin(\frac{1}{2}\,alpha\_0)^2}}\right)$$

```
> plot( subs(alpha_0 = Pi, t -> ang(t)), t = 0..1, y = -0.1..3.14);
> anglediff := ang - alpha_0*t:
> evalf(minimize(subs(alpha_0 = Pi, t -> anglediff(t)), t = 0..1));
```
$$-.1422292715$$
```
> evalf(maximize(subs(alpha_0 = Pi, t -> anglediff(t)), t = 0..1));
```
$$.1422292755$$
```
> pitch := simplify(-2*fe/sin(ang/2));
```
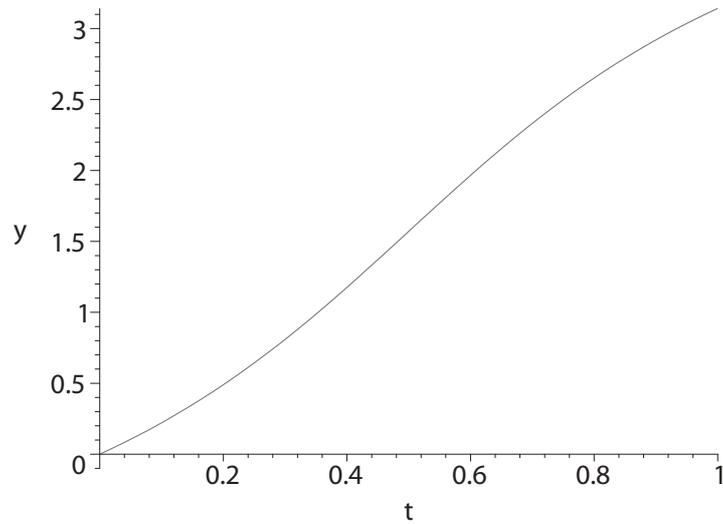
$$pitch := t^2 \, alpha\_e \, \sin(\tfrac{1}{2} \, alpha\_0) \, (-\%1 - t + t \, \%1) \Big/ ($$

$$(-1 + 2\,t - 2\,t\,\%1 - 2\,t^2 + 2\,t^2\,\%1) \sqrt{1 - 2\,t + 2\,t\,\%1 + 2\,t^2 - 2\,t^2\,\%1}$$

$$\sqrt{\frac{t^2\,(-1 + \%1^2)}{-1 + 2\,t - 2\,t\,\%1 - 2\,t^2 + 2\,t^2\,\%1}})$$

$$\%1 := \cos(\tfrac{1}{2} \, alpha\_0)$$

```
> plot( subs(alpha_0 = Pi, alpha_e=1, t -> pitch(t)), t = 0..1, y =
>  -1..1);
```
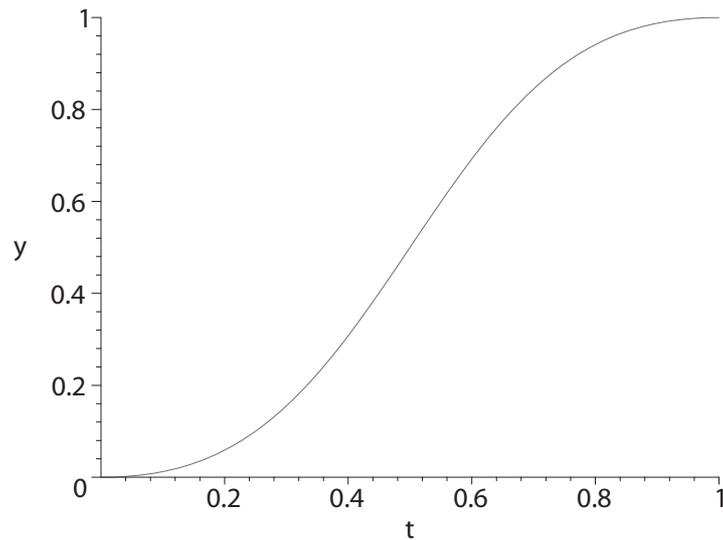


```
> pitchdiff := pitch - alpha_e*t:
```

```
>   evalf(minimize(subs(alpha_0 = Pi, alpha_e=1, t -> pitchdiff(t)), t =
>   0..1));
```
$$-.1501415529$$
```
>   evalf(maximize(subs(alpha_0 = Pi, alpha_e=1, t -> pitchdiff(t)), t =
>   0..1));
```
$$.1501415529$$

A.2 Log-matrix Blending Is Not Constant Speed

The constant speed property of log-matrix blending [3] has an interesting history. The author of log-matrix blending claims, without proof, that his method is not constant speed. Subsequently, a critique is posted on-line [13], which points out mistakes in the log-matrix blending paper [3]. Among several good insights, it unfortunately also mentions the fact that log-matrix blending actually *is* constant speed. This is not true, as we prove in this section.

A.2.1 Background on Log-matrix Blending

Before we start with the actual proof, we review the log-matrix blending method with a special focus on blending of rotations. This will be advantageous for the subsequent discussion.

Let us consider a simple situation of two 3×3 rotation matrices R_0 and R_1. Let R_t be an interpolation between those two matrices, i.e., a matrix which for $t = 0$ becomes R_0, for $t = 1$ becomes R_1 and for $0 < t < 1$ is a valid rotation matrix. What we informally referred as *speed* in the above, is actually an angular velocity of R_t. The formula expressing angular velocity in the body (moving) coordinate system is

$$M(\omega_t) = R_t^{-1} \frac{\partial R_t}{\partial t} \tag{A.1}$$

see [101]. Alternatively, we could also use a similar formula for angular velocity expressed in the spatial coordinate system. This angular velocity differs only by the reference coordinate system, and thus its magnitude (which we aim to compute) is the same. In our analysis, we will work with the body angular velocity, although we could equally well work with the spatial angular velocity. In Formula (A.1), $M(\mathbf{a})$ is a function mapping vector $\mathbf{a} = (a_1, a_2, a_3)$ to an anti-symmetric matrix

$$M(\mathbf{a}) = \begin{pmatrix} 0 & -a_3 & a_2 \\ a_3 & 0 & -a_1 \\ -a_2 & a_1 & 0 \end{pmatrix} \tag{A.2}$$

The multiplication of any vector $\mathbf{x} = (x_1, x_2, x_3)$ by this matrix corresponds to the cross product, i.e., $M(\mathbf{a})\mathbf{x} = \mathbf{a} \times \mathbf{x}$. Therefore, the vector ω_t in Formula (A.1) is the common

vector representation of angular velocity. The anti-symmetric matrix $M(\mathbf{a})$ is connected with matrix logarithms. If R is a rotation about axis $\mathbf{a}/\|\mathbf{a}\|$ with angle $\|\mathbf{a}\|$, then its logarithm $\log R = M(\mathbf{a})$. This gives us an intuitive explanation of the rotation matrix logarithm. To verify this fact, consider a differential equation describing rotation of a point $\mathbf{p}$ at time t with angular velocity $\mathbf{a}$:

$$\frac{\partial \mathbf{p}(t)}{\partial t} = \mathbf{a} \times \mathbf{p}(t) = M(\mathbf{a})\mathbf{p}(t)$$

The solution of this differential equation can be expressed using the matrix exponential

$$\mathbf{p}(t) = \exp(M(\mathbf{a})t)\mathbf{p}(0) \tag{A.3}$$

where $\mathbf{p}(0)$ is the initial condition, i.e., the position of the point at time 0. We can observe that the term $\exp(M(\mathbf{a})t)$ in Formula (A.3) is nothing but the matrix of rotation about axis $\mathbf{a}/\|\mathbf{a}\|$ with angle $t\|\mathbf{a}\|$. Therefore, the matrix R describing rotation about axis $\mathbf{a}/\|\mathbf{a}\|$ with angle $\|\mathbf{a}\|$ can be written as $R = \exp(M(\mathbf{a}))$. From this, it immediately follows that $\log R = M(\mathbf{a})$, as we wanted to show.

In the following, R_t will denote the result of log-matrix blending, given according to [3] as

$$R_t = \exp((1 - t)\log R_0 + t\log R_1) \tag{A.4}$$

where exp and log denote the matrix exponential and logarithm. The geometrical interpretation of matrix logarithm gives us an insight into what the log-matrix blending (limited to rotations) actually does: linear blending of the axis-angle representation of rotations.

A.2.2   Log-Matrix Blending in Maple

To show that log-matrix blending is not constant speed, it is sufficient to find two rotation matrices R_0, R_1, and show that the magnitude of angular velocity of their blend R_t, i.e., $\|\omega_t\|$ according to Formula (A.1), is not a constant function. Let us define R_0 as a rotation about axis $(1, 0, 0)$ with angle 1 radian, and matrix R_1 as a rotation about axis $(1/\sqrt{2}, 1/\sqrt{2}, 0)$ with angle $\sqrt{2}$ radians. This choice simplifies the following computations. The logarithms of R_0 and R_1 are

$$\log R_0 = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{pmatrix}, \quad \log R_1 = \begin{pmatrix} 0 & 0 & 1 \\ 0 & 0 & -1 \\ -1 & 1 & 0 \end{pmatrix}$$

and therefore

$$(1 - t)\log R_0 + t\log R_1 = \begin{pmatrix} 0 & 0 & t \\ 0 & 0 & -1 \\ -t & 1 & 0 \end{pmatrix}$$

We denote this matrix as $L_t := (1 - t)\log R_0 + t\log R_1$. The next step is to compute the exponential of matrix L_t. This can be easily done in Maple (it would also be possible to

use the Rodriguez formula [101], but the equations quickly become awkward for manual derivations).

```
>   with(linalg):
>   Lt := matrix(3,3,[0,0,t,0,0,-1,-t,1,0]);
```

$$Lt := \begin{bmatrix} 0 & 0 & t \\ 0 & 0 & -1 \\ -t & 1 & 0 \end{bmatrix}$$

```
>   E := simplify(evalm(exponential(Lt)));
```

$$E := \begin{bmatrix} \dfrac{1}{2}\dfrac{2+t^2\,\%2+t^2\,\%1}{1+t^2} & -\dfrac{1}{2}\dfrac{t\,(\%2+\%1-2)}{1+t^2} & \dfrac{1}{2}\dfrac{t\,(-\%2+\%1)}{\sqrt{-1-t^2}} \\[2ex] -\dfrac{1}{2}\dfrac{t\,(\%2+\%1-2)}{1+t^2} & \dfrac{1}{2}\dfrac{2\,t^2+\%2+\%1}{1+t^2} & -\dfrac{1}{2}\dfrac{-\%2+\%1}{\sqrt{-1-t^2}} \\[2ex] \dfrac{1}{2}\dfrac{t\,(\%2-\%1)}{\sqrt{-1-t^2}} & -\dfrac{1}{2}\dfrac{\%2-\%1}{\sqrt{-1-t^2}} & \dfrac{1}{2}\,\%2+\dfrac{1}{2}\,\%1 \end{bmatrix}$$

$$\%1 := e^{(\sqrt{-1-t^2})}$$
$$\%2 := e^{(-\sqrt{-1-t^2})}$$

```
>   Esubs := map(x -> subs( sqrt(-1-t^2) = i*B, 1/sqrt(-1-t^2) = 1/(i*B),
>   x), E);
```

$$Esubs := \begin{bmatrix} \dfrac{1}{2}\dfrac{2+t^2\,e^{(-iB)}+t^2\,e^{(iB)}}{1+t^2} & -\dfrac{1}{2}\dfrac{t\,(e^{(-iB)}+e^{(iB)}-2)}{1+t^2} & \dfrac{1}{2}\dfrac{t\,(-e^{(-iB)}+e^{(iB)})}{iB} \\[2ex] -\dfrac{1}{2}\dfrac{t\,(e^{(-iB)}+e^{(iB)}-2)}{1+t^2} & \dfrac{1}{2}\dfrac{2\,t^2+e^{(-iB)}+e^{(iB)}}{1+t^2} & -\dfrac{1}{2}\dfrac{-e^{(-iB)}+e^{(iB)}}{iB} \\[2ex] \dfrac{1}{2}\dfrac{t\,(e^{(-iB)}-e^{(iB)})}{iB} & -\dfrac{1}{2}\dfrac{e^{(-iB)}-e^{(iB)}}{iB} & \dfrac{1}{2}\,e^{(-iB)}+\dfrac{1}{2}\,e^{(iB)} \end{bmatrix}$$

```
>   Esubs2 := simplify(map(x -> subs( exp(i*B) = cos(B) + i*sin(B),
>   exp(-i*B) = cos(B) - i*sin(B), x), Esubs));
```

$$Esubs2 := \begin{bmatrix} \dfrac{1+t^2\cos(B)}{1+t^2} & -\dfrac{t\,(\cos(B)-1)}{1+t^2} & \dfrac{t\sin(B)}{B} \\[2ex] -\dfrac{t\,(\cos(B)-1)}{1+t^2} & \dfrac{t^2+\cos(B)}{1+t^2} & -\dfrac{\sin(B)}{B} \\[2ex] -\dfrac{t\sin(B)}{B} & \dfrac{\sin(B)}{B} & \cos(B) \end{bmatrix}$$

```
>   Efinal := map(x -> subs( B=sqrt(1+t*t), x), Esubs2);
```

$$
Efinal := \begin{bmatrix}
\dfrac{1 + t^2\,\%1}{1 + t^2} & -\dfrac{t\,(\%1 - 1)}{1 + t^2} & \dfrac{t\,\%2}{\sqrt{1 + t^2}} \\[3ex]
-\dfrac{t\,(\%1 - 1)}{1 + t^2} & \dfrac{t^2 + \%1}{1 + t^2} & -\dfrac{\%2}{\sqrt{1 + t^2}} \\[3ex]
-\dfrac{t\,\%2}{\sqrt{1 + t^2}} & \dfrac{\%2}{\sqrt{1 + t^2}} & \%1
\end{bmatrix}
$$

$$\%1 := \cos(\sqrt{1 + t^2})$$
$$\%2 := \sin(\sqrt{1 + t^2})$$

We had to help Maple with the substitution in order to obtain a real matrix (instead of complex). The result of log-matrix blending therefore is $R_t := \exp L_t$, where

$$
\exp L_t = \begin{pmatrix}
\dfrac{1 + t^2\cos(\sqrt{1+t^2})}{1+t^2} & -\dfrac{t\left(\cos(\sqrt{1+t^2})-1\right)}{1+t^2} & \dfrac{t\sin(\sqrt{1+t^2})}{\sqrt{1+t^2}} \\[3ex]
-\dfrac{t\left(\cos(\sqrt{1+t^2})-1\right)}{1+t^2} & \dfrac{t^2+\cos(\sqrt{1+t^2})}{1+t^2} & -\dfrac{\sin(\sqrt{1+t^2})}{\sqrt{1+t^2}} \\[3ex]
-\dfrac{t\sin(\sqrt{1+t^2})}{\sqrt{1+t^2}} & \dfrac{\sin(\sqrt{1+t^2})}{\sqrt{1+t^2}} & \cos(\sqrt{1 + t^2})
\end{pmatrix}
$$

Now we compute the inverse and derivative of R_t (`Efinal`), whose multiplication gives the angular velocity matrix $M(\omega_t)$, according to Formula (A.1).

```
>  dEfinal := map(x->diff(x,t), Efinal):
>  iEfinal := inverse(Efinal):
>  AngVelMat := simplify(evalm(iEfinal&*dEfinal));
```

$$
AngVelMat := \begin{bmatrix}
0 & -\dfrac{\cos(\sqrt{1 + t^2}) - 1}{1 + t^2} & \dfrac{\sqrt{1 + t^2}\,t^2 + \%1}{(1 + t^2)^{(3/2)}} \\[3ex]
\dfrac{\cos(\sqrt{1 + t^2}) - 1}{1 + t^2} & 0 & \dfrac{(-\sqrt{1 + t^2} + \%1)\,t}{(1 + t^2)^{(3/2)}} \\[3ex]
-\dfrac{t^4 + t^2 + \%1\,\sqrt{1 + t^2}}{(1 + t^2)^2} & -\dfrac{(-t^2 - 1 + \%1\,\sqrt{1 + t^2})\,t}{(1 + t^2)^2} & 0
\end{bmatrix}
$$

$$\%1 := \sin(\sqrt{1 + t^2})$$

```
>  simplify(evalm(AngVelMat + transpose(AngVelMat)));
```

$$
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0
\end{bmatrix}
$$

The last command verifies that `AngVelMat` is indeed an anti-symmetric matrix, as expected, and therefore has the structure from Formula (A.2). Now we extract the angular velocity vector ω according to Formula (A.2), and compute its norm:

```
>  omegat := matrix(1,3,[AngVelMat[3,2], -AngVelMat[3,1],
>  AngVelMat[2,1]]):
```

```
>   omegatLen := simplify(sqrt( AngVelMat[3,2]^2 + AngVelMat[3,1]^2 +
>   AngVelMat[2,1]^2 ));
```

$$omegatLen := \sqrt{-\frac{-t^4 - t^2 - 2 + 2\cos(\sqrt{1 + t^2})}{(1 + t^2)^2}}$$

Obviously, the function ω_t is not constant for $t \in [0, 1]$, see also its graph in Figure A.1.
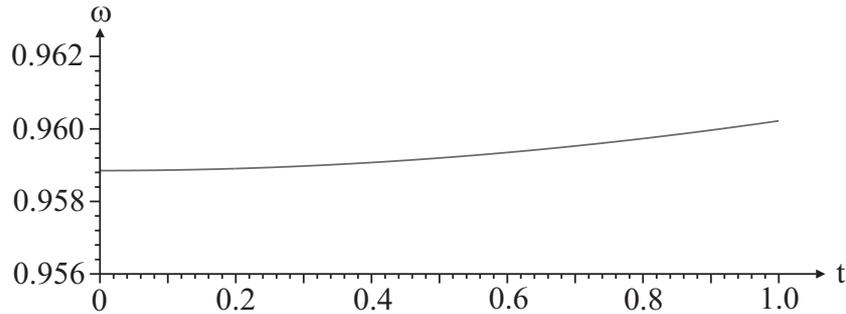

Figure A.1: Graph of $\|\omega_t\|$ for $t \in [0, 1]$

Although we have just shown that the speed of log-matrix blending is not constant, we observe that it is actually not far from constant. This is probably what accounts for the confusion of Bloom et al. [13] because, with numerical calculations, the slight non-constantness could be incorrectly explained as a numerical error. The observation that the speed of log-matrix blending is almost constant is in accordance with Alexa's statement that the blending is visually pleasing [3] (as motion with large variations of angular acceleration would not look pleasing). An interesting future work would be to find an upper bound of the angular acceleration of log-matrix blending in general.

B Acronyms and Symbols

AABB Axis Aligned Bounding Box

BVH Bounding Volumes Hierarchy

CD Collision Detection

CH Convex Hull

DIB Dual quaternion Iterative Blending

DLB Dual quaternion Linear Blending

DOP Discrete Oriented Polytope

FFD Free Form Deformation

FLOPS Floating Point Operations

FPS Frames Per Second

GPU Graphics Processing Unit

LBS Linear Blend Skinning

OBB Oriented Bounding Box

PCA Principal Component Analysis

QLB Quaternion Linear Blending

SAD Skinning Arbitrary Deformations

SBS Spherical Blend Skinning

ScLERP Screw Linear Interpolation

SLERP Spherical Linear Interpolation

SMA Skinned Mesh Animations

SVD Singular Value Decomposition