DIPLOMARBEIT

SKELETAL STRUCTURE GENERATION FOR OPTICAL MOTION CAPTURE

ausgeführt am Institut für
Softwaretechnologie und Interaktive Systeme
der Technischen Universität Wien

unter der Anleitung von
Univ.Ass. Mag.rer.nat. Dr.techn. Hannes Kaufmann

durch

Christian Schönauer
Lorenz-Müller-Gasse 1/4416
1200 Wien

Wien, am 12. Dezember 2007

Abstract

Motion capture systems today have to deliver high quality motion data, while being flexible and easily adaptable to different actors. Therefore, accurately determining parameters of a subject's skeletal structure is crucial. Inferring these values automatically from optical motion capture data without additional measurements, however, is a challenging task. This thesis describes the steps necessary to calculate the joint positions and limb lengths using data from a passive optical tracking system.

The algorithm is a multi-stage process that includes the tasks of automatic marker labeling, limb-wise clustering of markers and calculation of joint positions. Finally an estimate of the topology and the parameters of the articulated structure are computed. Since the topology is inferred from the data, no model has to exist in advance. This in turn makes the implemented system flexible enough to capture not only human motions, but motions of an arbitrary articulated structure, without any adaptations or additional effort. The core functionality of the system, which is the skeleton fitting task, is done using a distance function, that is applied to marker positions. This function then is minimized by a non-linear minimization algorithm.

Tests of the system have been performed with artificially generated data and a construction of rods linked with articulations. The results show high accuracy for the artificial data. For the tracked data sets also satisfactory outcome is produced.

Zusammenfassung

Motion Capture Systeme müssen heutzutage hochqualitative Bewegungsdaten liefern. Trotzdem sollen sie flexibel sein und leicht für verschiedene Darsteller anzupassen. Deswegen ist die Bestimmung der Parameter des Skeletts ein wichtiger Bestandteil solcher Systeme. Die automatische Bestimmung dieser Werte unter Verwendung von optischen Bewegungsdaten ohne Zuhilfenahme zusätzlicher Messungen ist jedoch eine anspruchsvolle Aufgabe. Diese Arbeit beschreibt welche Schritte notwendig sind um die Gelenkspositionen sowie die Länge der Gliedmaßen zu bestimmen. Dazu werden die Daten eines passiven optischen Tracking Systems verwendet. Der verwendete Algorithmus ist ein mehrstufiger Prozess. Zunächst werden die getrackten Markerpositionen den physikalischen Markern zugeordnet und nach Körperteilen gruppiert. Dann werden die Gelenkspositionen und die Topologie des Skeletts bestimmt. Schließlich werden die Parameter der Struktur berechnet. Da die Topologie ausschließlich aus den Daten bemessen wird ist im Vorhinein kein Modell notwendig. Dieser Umstand macht es möglich mit dem implementierten System nicht nur menschliche Bewegungen aufzuzeichnen. Jede beliebige durch Gelenke verbundene Struktur kann damit ohne jegliche Adaptierung oder zusätzlichen Aufwand erfasst werden. Die Kernfunktionalität der Applikation - die Berechnung des Skeletts - wird mit Hilfe einer Distanzfunktion bewältigt. Diese wird auf die Markerpositionen angewendet und durch einen nichtlinearen Optimierungsalgorithmus minimiert.

Das System wurde mit verschiedenen Daten getestet. Neben menschlichen Bewegungdaten wurden generiertes Datenmaterial und Messungen einer Konstruktion von durch Gelenke verbundenen Holzstäben verwendet. Die Resultate zeigen hohe Genauigkeit für die künstlichen Daten. Für die getrackten Datensätze wurden ebenfalls zufriedenstellende Ergebnisse erzielt.

Acknowledgements

First of all I want to thank my supervisor Hannes Kaufmann for his guidance and for always having encouraging words for me.

I thank Thomas Pintaric for his help with the *iotracker* and the input data.

Many thanks go to Christine, my brother Robert, Dominik and other friends for manifold support and for distracting me from work, both in the right dosage.

Last but not least I want to thank my parents for supporting me throughout the years of my studies and for always encouraging me. Special thank goes to my mother for putting her sewing skills into service to build the first MoCap suit.

Contents

Chapter 1

Introduction

1.1 Motion Capture

Throughout the literature many different definitions of motion capture exist. Many of them include statements like "... must now take on the challenge of defining what we mean by the phrase *motion capture...*" [Fur99] or "Deciding on which term to employ is one of the challenges..." [JFGT02]. From that it is easy to see that defining the term motion capture is a rather difficult task. Instead of discussing on several pages what motion capture might or might not be I will simply take the definition from [JFGT02]. It states: "Motion capture is the recording of a manipulable representation of a motion from sensing that motion." The result of the motion capture process then is called motion data. In our case the manipulable representation would be the parameterized skeleton. The motion data includes its transformations at any point within the considered time-span. In general, however, the form of representation used is dependent on the application.

A very common abbreviation of motion capture is MoCap, which I will frequently use throughout the rest of this work.

The process of inferring position and/or orientation data from sensor data is called tracking. Tracking is a substantial part of motion capture and good quality of tracking data is crucial for motion capture. All kinds of physical phenomena are used for tracking depending on the application. The motion data in our system is derived from images taken by multiple cameras using an optical tracking system. For a more detailed description of tracking systems please refer to chapter 2.

The process of motion capture in this context does not necessarily include animation as sometimes is suggested in the literature. It is merely the extraction of motion data. This data then can be used to create animation in a process most commonly called performance animation, although many other terms exist as well. Since there are other applications for motion capture than animation, strictly separating these terms makes sense. So MoCap can be used for performance analysis in sports, biomedical analysis or virtual reality as will described

in more details in the next section.

MoCap systems for performance animation can be classified into two kinds of systems. On the one hand there are online-systems, which can be used to create animation from motion data in real-time. With the improvement of tracking technologies and computing performance these become more and more common. On the other hand there are offline systems, where motion data has to be processed before the animation is started. Our system can only be used offline for reasons that will become apparent later in this work.

1.2 Areas of Application for Motion Capture

Motion capture has a broad variety of applications. The most prominent area is of course the entertainment industry, which needs realistic motion data for animation movies and computer games. However, some other fields of application exist, where captured data can be used to analyze motion. These areas of application are medicine, sport and virtual reality. The next sections will describe these fields. Here an emphasis is placed on the entertainment sector, since our system is intended primarily for performance animation, while it might not be suited for other fields.

1.2.1 Medicine

Motion capture technologies are nowadays often used for clinical purposes. Many systems have been developed to produce biomechanical analysis based on motion data. These systems are used in sports medicine, rehabilitation and neurology to identify anomalies in the movement of single limbs or the whole body. The result of the analysis can then be used to better diagnose the patient and determine the best treatment. Especially gait analysis is a common application for motion capture and can help to identify the causes of walking abnormalities [Mot07a].

1.2.2 Sport

Motion capture systems offer an opportunity to analyze the motions of an athlete, that otherwise wouldn't be possible. So, fast motions can be captured, split up into their different components and analyzed from different perspectives. The findings then can be used by an athlete to develop better training methods and improve performance.

1.2.3 Virtual Reality

Real-time motion capture allows a user to control an avatar (a virtual representation of the user) in a virtual environment. This allows natural interaction with the virtual world instead of the limited possibilities of the available input devices (space mouse etc.)

1.2.4 Entertainment

The entertainment sector is the biggest application area for MoCap. Motion data is here used for performance animation in movies on the one hand and for animation in computer games on the other hand. Since our system is intended for performance animation these two fields are described in more detail.

1.2.5 Movies and Television

More and more movies contain or are entirely made by animation. This on the one hand makes the production of movies cheaper, due to for example the use of animated background characters. On the other hand animation enables breathtaking effects and stunts that wouldn't be possible otherwise. So in a dangerous scene the actor can be simply replaced by a virtual character, which is animated using the actors motion data. In this case of course the virtual character has to optically match the actor as much as possible to make the illusion work.

In general, however, motion data is independent from the captured subjects exterior it can be used to animate any humanoid character. One stunning example for the use of performance animation is the movie "The Polar Express" . In this movie the performance of one actor (Tom Hanks) is used to animate several characters among them a little boy. This without motion capture hadn't been possible. Although according to [Gor06] the performance animation was a lot of effort and added a lot to the projects total cost of 165 Million $ and three years of work, without it the movie would have been even more expensive. Figure 1.1 shows Tom Hanks wearing a MoCap suit and the corresponding scene from "The Polar Express".

Even fast motions like skateboard-tricks can be captured using MoCap. So in the animated movie "Boom Boom Sabotage" staring Tony Hawks 95% of the animations are done using MoCap according to [Vic07a]. Here not only the actors motions but also the flipping and spinning skateboards had to be captured.

In my opinion one of the best examples for successful use of performance animation in movies is "Lord of the Rings". The seamless integration of the virtual character Gollum in a real movie is just stunning.

1.2.6 Computer games

With the increase of graphical details in computer games realistic animation of virtual characters became more and more important within the years. Due to the harsh competition between the game development studios to produce the most realistic games the computer game industry became the largest market for motion data([Men99]).

According to [Mot07b] two areas of application of motion data exist in computer games. On the one hand there is the real-time playback and on the other hand the so-called cinematics.

Figure 1.1: Tom Hanks and one of his virtual counterparts in "Polar Express" (Courtesy Warner Bros. Pictures)

For real-time playback all motions and according actions that a virtual character should be able to perform are recorded using MoCap. Depending on the players input these motions are then replayed during the game.

Cinematics are pre-rendered movie-clips that are intended to introduce the player to a plot and develop the story of the game. They are used to make games more immersive and to tie parts of the games together. Cinematics are usually not done by the game developers themselves but outsourced to specialized studios. Therefore they are not much different from a traditional movie production. Within the last years,however, where graphical details have increased within the games themselves it can be observed that game developers decide against cinematics in a classical sense. Instead the game's graphics engine is used to produce animations, which then are composed with voice recordings to produce some kind of cinematics.

Chapter 2

Related Work

2.1 Classification of Tracking Systems

The following subsections present properties of trackers that can be used to categorize and evaluate a system. However, it is impossible to give a general standard for a "good" tracking system by these values. This is because it ultimately depends on the intended area of application if a tracker is adequate.

2.1.1 Outside-In/Inside-Out/Inside-In

Before the tracking process can be started a reference coordinate system has to be defined, so the tracked object (target) can be assigned coordinates. The coordinate system can be relative to the sensors or the emitters/transmitters. Depending on this relation and whether the devices are placed on the target or in the environment three types of tracking systems can be identified.

Inside-In

For inside-in systems the sensors and emitters as well as the reference coordinate system is placed on the target. This has the advantage of limiting influences from the environment. Also the position/orientation of the limbs relative to each other can be calculated very fast and precise. On the other hand no global position/orientation can be calculated since no relation to an outside reference point exists. Examples for inside-in systems are electromechanical and inertial systems as described. in 2.2.

Outside-In

Systems where the sensors are on fixed reference points in the environment and the emitters are fixated on the target are called outside-in. Since the coordinate system is aligned with the sensors in the environment the absolute position/orientation of the target can be calculated. Another advantage is that for

example optical emitters are rather small compared to the sensors, which results in comfortable body suits for motion capture applications.

Inside-Out

When the sensors are located on the tracked object, the emitter/transmitter is installed in the vicinity and the coordinate system is aligned with the latter we are talking about inside-out systems. This kind of system has the advantage that the absolute position/orientation of the emitters and thus of the tracked object is known. The main drawback of this approach however is that the data from the sensors has to be somehow transferred to the PC for processing, which requires wiring of the tracked object or a high bandwidth wireless transmission technology.

2.1.2 Properties of Tracking System

For all tracking technologies some properties, like latency, accuracy and update rate can be measured. These characteristics will be used in the next sections to evaluate the discussed tracking systems. The properties are described shortly in the following sections and are discussed in more details in [BC03], [RDB01], and [WF02].

Latency

The time between the change of an object's orientation/position and the detection of the change by the sensor/acquisition-subsystem is called latency. For realtime-tracking it is required to be as small as possible, because otherwise the time delay between movements and their display might be unacceptable. If the motion is recorded only latency is less crucial as long as the update rate is sufficient.

Update Rate

The update rate of a tracker specifies the number of measurements that the tracker outputs every second. For an optical tracking system for example this would be one divided by the time necessary to process an image and extract the 3D position. The higher the update rate the better the precision of the captured motions (i.e. less interpolation is necessary for the animation process), which especially accounts for fast motions.

Phase Lag

The time interval from the change of an object until it is outputted by the tracker is called phase lag.

Accuracy

Accuracy is usually separated into static and dynamic accuracy. Static accuracy is the maximum deviation of the tracker's output from the actual value, when the position/orientation of the tracked object is constant. Dynamic accuracy measures the accuracy for a moving object and is dependent on the static accuracy. For most tracking technologies the accuracy is a non-constant parameter, but dependent on the distance between emitter and sensor.

Drift

The drift represents the increase of tracking error over time. This is especially a problem for inertial trackers, as we will see later. For stationary objects drift is also called stability or creep in the literature. When the error resulting from drift grows too large the tracker has to be recalibrated.

Jitter

Jitter specifies the changes in the tracker output for an object, that has constant position/orientation. While a constant error might be less noticeable for an observer, large jitter results in tremor an unsteady motions.

2.2 Tracking Hardware/Technologies for whole Body Motion Capture

Currently three technologies are frequently used for full body MoCap, which will be described in this section.

2.2.1 Electromechanical

Electromechanical tracking systems offer a simple and fast way to obtain the pose and movements of a human user. For the tracking process an artificial skeleton, made of rigid segments connected with articulations, is built around the body. Since it is outside the body it is called Exoskeleton. This skeleton is fitted in a way, that the rigid parts match the limbs and the articulations coincide with the joints of the human body as well as possible. The better the articulated skeleton suits the user, the more accurate are the captured motions. Therefore most systems can be adapted to different body sizes. Nevertheless it can hardly be avoided, that the user will be restrained in his movements.

When the user moves his limbs the Exoskeleton follows and in the ideal case the articulations obtain the same angles as the joints. The angles of the artificial joints are then being calculated using measurements of goniometers. Often electromechanical transducers such as potentiometers, which have been used for electronic products of all sorts for many years, are used for this task. The measurements of the transducers are then transferred to a computer using cables or

in more recent systems wireless connections. From these readings, the pose of the exoskeleton is calculated and the position and orientation of the users limbs can be estimated.

As an alternative to goniometers, bend sensors can be used to retrieve the angle. Fiber-optic sensors, resistive ink sensors and others are described in more detail in [BKL04]. Another possibility would be to use two gravito inertial sensors as a virtual goniometer. No matter which technique is used, the orientation of the two limbs, adjacent to the joint, to each other is obtained. Now that the relative orientations are known, forward kinematics can be used to calculate the pose of the body using a hierarchical model of the body. Starting from a root (usually the torso or pelvis) the angles are used to find the positions of all limbs. The method is pretty simple - compared to other methods - and can even easily be done in real time.

The problem, however, as with all inside-in systems, is that an absolute orientation and position cannot be obtained. Therefore, additional tracking techniques, as suggested in [Sta02], can be used or one of the following methods applied.

One possibility to get ground truth measurements is to simply connect the exoskeleton to a fixed reference point by additional linkages. In case the link provides at least 6 DOF the user can roam freely within the (very limited) range, while position and orientation can be calculated. Again angular measurements and forward kinematics are used to derive the sought-after values. Besides the obvious limitations this method has the disadvantage of rendering multi user applications nearly impossible.

Another method of obtaining the absolute position and orientation is described in [Sta02]. It only uses the data provided by the goniometers and relies on the fact that the position of one foot relative to the other can always be calculated using forward kinematics. Starting with one foot at a reference point, the position of the other foot is inferred and , as soon as it is put on the ground used as new reference point. Although no implementation is known it should work under certain conditions. Most important of all, it has to be ensured, that a foot doesn't move once it touches the floor, which of course isn't true for normal movement. The big problem with this method, however, is that measurement errors propagate from step to step. Thus, sooner or later a virtual character, animated with this motion capture data, would face in a totally wrong direction and probably either float or walk below the floor.

Taking aside the positioning problem and the fact that the exoskeleton hinders the user in his movements electromechanical tracking suffers from another serious drawback: It is not possible with current commercially available systems to track anything else than the human body. Although not necessary for many applications, handheld objects or animals for example can't be tracked.

Electromechanical tracking, however, has some advantages that makes it superior to other tracking technologies for some applications. The first and probably most important reason is, that it is cheap, fast and easy to use. Almost no post-

processing and very little computational power is needed, which makes its use in an real time environment possible. Additionally an exoskeleton is an inside-in system, which makes it independent of influences from the surroundings. So the light conditions in the capture area as well as potential magnetic interferences or ferromagnetic materials are of no concern. Another advantage of electromechanical systems is their large capture volume. With wireless systems the range within which the pose can be estimated is only limited by the range of the used transmission technology. One producer even boasts, that his electromechanical system is having the largest capture range of any motion capture systems ever built. This, however, is only true as long as no other tracking technique is used to obtain the absolute orientation and position. From this additional method also depends whether or not the system can be used for tracking multiple subjects. As mentioned before, a telescope arm fixated on the back of each user doesn't really help to "socialize".

2.2.2 Magnetic Tracking

Magnetic tracking technologies have a long history, being introduced in the mid-seventies and, despite the fact that there are lots of other systems available, still play a vital role in motion capture. They are using the fact, that sensors can infer position and orientation information from magnetic fields due to a phenomenon called magnetic coupling. If only the 2D orientation is to be inferred the earth‘s magnetic field can be used, as it is in a compass. For motion capture, however, the three-dimensional position and orientation of the captured object‘s limbs are needed. For that purpose at least three magnetic fields are required and thus have to be generated artificially. The transmitter, that generates those fields, is at a fixed location, while the sensors are usually placed on the user‘s limbs. Magnetic tracking systems thus are by definition inside-out systems.

The magnetic dipole fields, necessary for the tracking process, are generated by circulating an electric current in coils of wire. In the transmitter three coils are assembled orthogonally to each other, which are excited sequentially to generate three orthogonal magnetic fields. To excite the coils AC and DC currents are being used, generating an oscillating magnetic field in the first case and a pulsed in the second. Depending on the source used, the system has specific properties I will describe in detail later in this chapter. The magnetic fields, generated by the transmitter, induce a magnetic flux in the receiver. The three sensors in the receiver measure the components of the flux. From these measurements the system can infer the position and orientation of the receiver. This is done by an electronic unit, which then transmits the data to a computer for further processing.

Electromagnetic tracking systems have some advantages over other technologies, which have made them the most widely deployed for motion tracking. One of the most important merits is, that they have no line of sight restrictions. Due to the fact that human limbs as well as other non-metallic objects have no effect

on magnetic fields, the user of the system doesn't have to worry about the system loosing track of any body parts. Furthermore receivers of magnetic tracking systems are lightweight and very compact, which makes them easy to integrate into a motion capture suit. In addition, most systems have wireless versions, allowing the user to move unrestricted within the capture volume. Depending on the desired capture volume is the price of a magnetic tracking system, which can be relatively inexpensive compared to other systems, if only a short tracking range is required.

The working volume is limited by the magnetic field strength of the transmitter, which can not be arbitrarily increased, because of the possible negative side effects on the human body or electronic devices. It is therefore limited to a range of up to 5 meters around the transmitter. There is, however, a trade-off between the range, latency and resolution as pointed out at the end of the chapter. Yet the biggest drawback of electromagnetic trackers is, that external noise is created by metal objects or devices, such as CRT monitors. If they are located close to or within the tracking area, the magnetic field generated by the transmitter and thus the tracking data is disturbed. This error, however, is a static function of the position of the receiver, provided that the transmitter is fixed and the surrounding metal doesn't move. In a phase preceding the tracking process called "tracker calibration" the error can be measured and stored in a calibration table. From these measurements the position of the receiver can be corrected. For more details on the calibration process see [Kin05].

AC Magnetic Trackers

For AC trackers the three coils of the transmitter are fed with an alternating current of 7-14 kHz. The receiver of such a system contains three orthogonal coils, into which a current is induced by magnetic coupling. This current is proportional to the amplitude of the magnetic flux and the frequency of the oscillations. The induced voltages are then sampled by an electronics unit, which produces at least 9 values(3 sensor values in three excitation phases). In some systems oversampling is used at this point to get cleaner data.

The problem with AC magnetic trackers is that the magnetic fields generated by the transmitter induce eddy currents in metal objects. With the conductivity of these metals the induced current increases. The circulating eddy currents produce a magnetic field, that opposes the magnetic field of the transmitter. This in turn affects the accuracy of the system. It has been observed, that AC magnetic trackers are less influenced by ferromagnetic steel, copper, ferrite and mild steel than DC magnetic trackers. Furthermore, the earth‘s magnetic field as well as mains power wiring causes no interference. Brass, aluminium and stainless steel, however, produce more noise within an AC magnetic field. More details on that matter can be found in [Sta02] and [BC03].

DC Magnetic Trackers

In the transmitter of a DC magnetic tracking system one coil at a time is excited by a pulsed constant current. The receivers of a DC magnetic tracker consist of an assembly of either the outdated fluxgate magnetometers or solid state technologies, like hall effect sensors. They measure the induced voltages in the three phases during which the transmitter generates the orthogonal magnetic fields. In a forth time phase no magnetic field is generated. This pause is used by the receivers to measure the influence of the earth's magnetic field and other interferences. This error is subtracted from the values obtained in the other phases before the position and orientation is calculated.

DC systems are generally less influenced by eddy currents due to the fact that only the rising edge of the DC pulse creates them. A short delay between creating the magnetic field and sampling is usually enough to make sure the eddy currents die out, before measurements are taken. [BC03], however states that highly conductive materials like copper pose a problem even for DC magnetic trackers. Here the induced currents exit longer than the duration of the systems time delay.

A big problem with DC magnetic trackers is, that the sensors are sensitive to interferences in a low frequency band. Such interferences are generated by main power wiring and CRT monitors. [NMFP98] therefore suggests that the sampled voltages should be synchronized with the main power supply and sampled with twice the frequency. Averaging over two taps then cancels out the interference. This of course results in an increase of the latency.

Range, Latency and Resolution

In magnetic tracking systems exists a trade-off between the systems tracking range, its latency and resolution. The problem is, that the magnetic field strength falls off with the cube of the distance to the transmitter. Resolution even is proportional to the fourth power of the distance. This applies for the position as well as for the orientation, which has been shown experimentally. Since the magnetic field strength cannot be increased infinitely, as was pointed out earlier, the only chance to improve resolution with increasing distance to the transmitter is filtering. [BC03] tries to make this trade-off more vivid using the following calculation. Assuming that the resolution is sufficient at a distance r from the transmitter and we want the same resolution at distance $2r$. The noise at distance $2r$ is 16 times as high as at r. Using a rectangular filter we would need 256 taps to filter the noise back to the original level. This would increase the latency of the system by a factor of 128. This example exaggerates a bit, because with more sophisticated filters the number of taps needed can be reduced dramatically. Nevertheless it illustrates the problem of the trade-off very well.

2.3 Optical Tracking

Optical tracking systems use cameras to obtain position information. Therefore, objects/features are detected in the images taken by the cameras to infer their 3D coordinates. Most optical trackers are ouside-in systems, where the cameras are placed on fixed positions and the user moves within a capture space covered by the cameras. Inside-out systems, where the camera is fixated on the tracked object (e.g. the users head or head mounted display), can hardly be used for whole body motion capture and thus are not discussed here.

The coordinates of detected features are calculated within a world coordinate system, which is defined with respect to the positions of the cameras. Furthermore the world coordinate systems z-plane has to be aligned with the ground floor. This is usually done by localizing objects with known physical dimension and position in the camera images [GF05].

Most commercially available optical motion capture systems use CCD (charge coupled device) cameras for tracking. The number of cameras often is scalable, so the tracking area can be custom sized to the application. For the tracking process the cameras are placed in the environment providing different perspectives of the tracked subject. At discrete points in time the cameras then capture images from which the target‘s position information can be inferred. To ensure that the cameras all capture at the same time they have to be synchronized with high precision. This is important, because even a small deviation can result in large measurement errors.

For motion capture, usually systems with high frame rates are used, which results in a high effort for post-processing. Especially when multiple actors need to be tracked and many cameras are used to avoid occlusion, processing large amounts of data becomes an issue. This is even more crucial, when the motion capture process has to run in real-time, which makes massive parallel processing necessary. The required computational power in turn, along with the expenses for the other hardware components, makes optical tracking systems relatively expensive.

Optical tracking systems can be coarsely divided into two groups: marker-based systems and marker-less systems. These, along with their properties, will be discussed in the following sections

2.3.1 Marker-Based Optical Tracking Systems

Markers are objects that are placed on the actors body. The idea of using markers is that important parts of the body are decorated with landmarks (the markers) so they can easily be identified on the captured images. Markers either emit light (active markers) or reflect light (passive markers).

The markers are usually fixated on a tight fitting suit or for cheaper systems can be strapped on as presented in [Wei04] and used in the first version of our suit.

When working within the visible light spectrum problems arise with changing light conditions and optical noise. The solution to this problems is working with infrared light. This has the advantages that artificial lights can't cause disturbance and tracking even works in dark environments. In order to work in the infrared spectrum, cameras have to be equipped with infrared filters. Active markers in turn use infrared LEDs (light emitting diode), while for passive markers infrared spotlights are used.

Active Optical Systems

As mentioned before active markers emit light using infrared-LEDs or other technologies. This has the advantage that they appear very bright on the captured images. Also the identities of the markers can be relayed by having the markers blink with different frequencies. This reduces post-processing time compared to passive markers and ensures correct identification of markers, for example after they have been occluded for a longer time period. For the identification to work correctly, however, the markers have to be synchronized with the cameras, which creates an additional effort. Furthermore, active markers are relatively large and need a power supply. This can lead to a setup, where the actor is hindered by an uncomfortable suit and wires. Finally, active markers don‘t emit an equal amount of light into all directions. This can result in phenomenon, that markers are not recognized correctly even though they are within the camera‘s field of view.

Passive Optical Systems

For passive markers mostly spheres coated with retroreflective material are used. Also, although not very common, bright patches on a dark suit and other objects have been used in passive optical motion capture systems [GF05]. The main advantages of passive markers are that they are untethered and - for the spheres with some millimeters to centimeters in diameter - relatively small. This in turn makes it easy to incorporate the markers into body suits, which give the actors a maximum of freedom.

Passive markers reflect incoming light back into the direction of the source. Therefore (infrared)spotlights are placed in the direct vicinity of the cameras or integrated in the cameras casing. So markers appear as bright spots in the images captured by the cameras. During the alignment of the cameras and spotlights it has to be made sure, that the flashes of the spots don't blind cameras. For that reason usually the cameras are placed higher than the captured subject, so they can be oriented downwards.

The Properties of Marker-Based Optical Tracking Systems

With optical tracking systems good update rates can be expected. This is not only because the position information is transferred to the sensors(the cameras)

by the speed of light; also cameras can capture with up to 2000 frames per second [Vic07b]. To handle all the captured data the cameras often have built-in processing capabilities to detect marker positions in the images. These are then transferred via firewire or Ethernet to the PC. Using such systems, markers of only a few millimeters in diameter can be tracked with an accuracy in the sub-millimeter domain. The resolution, however decreases with the distance of the tracked object to the camera. Since the space between two points on the image plane becomes smaller with increasing distance, resolving them spatially becomes harder.

Another advantage of optical tracking systems is the freedom of movement within a large capture area. As has been mentioned this area is scalable by adding cameras. In today‘s systems dozens or (theoretically) hundreds of cameras can be combined to gather motion data.

Another merit is the capability of tracking multiple actors as well as objects of any kind at the same time. Furthermore commercially available systems are advertised to be easy to operate and calibrated within a few minutes [Pha07]. Vibrations, however, make occasional recalibration necessary to ensure high quality motion capture. Additionally portable systems exist that can be used in almost any (indoor) environment.

A major problem of optical tracking systems, however, is the occlusion of markers, which temporarily makes body parts invisible to the system. For example a squatting position can cover a lot of markers. Adding more cameras only helps to a certain extent to overcome this problem, letting aside that with additional cameras the effort of post-processing increases. So poses sometimes cannot be reconstructed correctly and for systems with passive markers identities might not always be reconstructed correctly.

Finally optical tracking systems are relatively expensive compared to other tracking technologies like mechanical tracking. For applications where high quality motion capture is needed it nevertheless often is the technology of choice.

2.4 Inertial Tracking

Inertial tracking systems use the physical phenomenon of mass inertia for calculating position and orientation information. Therefore, two types of sensors are needed: gyroscopes and accelerometers. Both kinds of sensors measure the rate at which they are moved. A gyroscope senses the angular velocity (i.e. the rate of change in orientation), while an accelerometer measures the rate of change in the translation velocity.

State-of-the-art rate-gyroscopes exploit the Coriolis effect using a vibrating resonator chip. The angle of orientation can then be determined through integration over time. To determine the three DOFs for orientation - yaw, pitch, and roll - three gyroscopes are needed and assembled orthogonally.

Accelerometers on the other hand determine position information. The rate

of change in translation velocity measured by the sensor has to be therefore integrated twice over time. Then, as long as the start position is known, the position in the world coordinate system can be calculated. To calculate the three-dimensional position three accelerometers are used and ideally aligned with the axes of the gyroscopes. Modern accelerometers use solid state technologies.

Inertial tracking systems like electromechanical systems are inside-in and therefore many possible problems, like line of sight issues or influences of the environment, are prevented in advance. Also no additional devices like cameras or magnetic field generators have to be placed in or around the capture area. Furthermore the range is only limited by the cables used to transfer the measurements to a PC or in more recent systems by the wireless transmission technology used. In addition the sensors are very small and thus can be easily attached to a body suit that doesn't hinder the actor in his movements. Finally inertial tracking systems have very little jitter, which according to [BC03] is additionally filtered through the integration. Therefore, without the need of filtering latency is very small.

The main problem with inertial tracking systems, however, is that accumulating errors - the so-called drift - create a deviation between real and measured motion that increases over time. While the error of gyroscopes increases proportional to the time with accelerometers the increase of error is proportional to the square of the time. For many applications these errors or a recalibration might not be acceptable. In [Sta02] the interested reader finds a more detailed discussion of drift rates of different inertial sensors.

2.5 The Tracker

The tracker used for our motion capture system is an *iotracker*, which is a passive marker-based infrared-optical motion-tracking system. It was primarily built for collaborative virtual reality and augmented reality applications, but is also well-suited for motion capture. The original intention was to provide an affordable system for smaller educational institutions like secondary schools. Therefore the components were selected to have good performance, while being relatively cheap. For that reason commodity hardware is used to minimize the cost. Also calculations are performed on PC-Workstations. A short overview over the trackers hard- and software is given in the following paragraphs, while a more detailed illustration is given in [PK07] and [Pin07].

iotracker relies on 4-8 FireFlyMV cameras from Point Grey Research. They produce monochrome images with a resolution of 640x480 at 60 frames/ second. Attached to the cameras are LED-arrays which generate near-IR strobe lights with a wavelength of below 850 nm. The cameras are synchronized with the strobes and are equipped with a band-pass filter to minimize distortion. From the cameras the pictures are transferred to a PC-Workstation for processing.

The software framework, which calculates the 3D positions from the images

is written entirely in C++ and supports parallel processing. Only for the camera calibration a `MATLAB` toolbox is used. For the intended areas of application the two main tasks, which are the feature segmentation and the projective reconstruction, have to be performed not only in real time but with a latency of below about 40 ms. For motion capture that wouldn't be necessary, however the immediate inspection of the results facilitates the capturing process. Also improvement of the program described in this thesis as well as parallelization might lead to a real time motion capture system.

The performance of *iotracker* might be slightly better when used for motion capture than depicted in [PK07] This is because there the targets (clusters of markers, with fixed distances to each other) have to be identified, which is not necessary for motion capture. However, the measured accuracy of $\pm 5mm$ and the jitter of less than $0.05mm$ and $0.02°$ (in the center of the capture area) remains the same.

2.6 Inverse Kinematics

Inverse kinematics (IK) is a technique that was originally developed for robotics applications. Later it was adopted for use in animation and motion capture and now plays a key role in these fields. IK is used to control the movement of linkages consisting of rigid parts connected by rotational and/or translational joints. In robotics this is usually a mechanical arm that is connected by some 1 DOF joints. The task of IK is now to set the rotations/translations of the joints in a way that the tip of the arm - the so-called end-effector - is placed at a desired position. In the following, however, only kinematic chains with rotational joints are considered, because translational joints are not needed for our intent.

For animation purposes hierarchical skeletons are modeled to resemble the real skeleton of the animated subject. Such a skeleton model can contain more than hundred DOFs, which are hard to control by an animator (if not performance animation is used). The skeleton, however, can be seen as a collection of linkages called kinematic chains and connected by a common root. These linkages each have their own end-effector(e.g. a hand) and therefore IK can be used to control the joint-angles (e.g. shoulder and elbow).

Another kinematics method is forward kinematics. As opposed to inverse kinematics, forward kinematics uses the known joint angles to find the position of the end-effector.

2.6.1 Problem Definition of Forward Kinematics

The problem of forward kinematics can be more mathematically defined using the following notation: $v = (q_1, .., q_n, t_1, ..., t_n)$ is a status vector containing the rotation parameters q_i for the ith joint and the translations resulting from the length of the ith section of the kinematic chain. Function $f(v)$ then returns the

position and orientation G of the end-effector. If q_i and t_i is given for every link then the resulting G is always unique.

2.6.2 Problem Definition of Inverse Kinematics

The inverse kinematics problem can be defined as finding the rotations q such that the given rotation and position G of the end-effector equals $f(v)$. In other words find q where

$$f(v) = \prod_{i=1}^{n} A_i(v_i) = G \text{ where } A_i(v_i) = \begin{bmatrix} R(q_i) & t_i \\ 0 & 1 \end{bmatrix} \text{ and } G,\ A_i(v_i) \in \mathbb{R}^{4x4}$$

$R(q_i)$ is a 3 by 3 rotation matrix created from the angular values stored in q_i. A_i is a 4 by 4 transformation matrix that for each link holds the rotation and translation. In case the inverse kinematics problem is solved correctly multiplying these transformation matrices starting at the end-effector results in the transformation matrix given by G.Ideally the angles q can be calculated by simply inverting function f to f^{-1}, which unfortunately most of the time is not possible.

The end-effector has 6 DOFs (3 rotational and 3 translational) depending on how much DOFs the joints of the kinematic chain are having altogether $DOF(q)$ the IK problem is said to be:

- well posed if $DOF(q) = 6$
- under-constrained if $DOF(q) > 6$ and
- over-constrained if $DOF(q) < 6$

2.6.3 Classification of IK Algorithms

IK Algorithms can be coarsely divided into *analytical* and *numerical* algorithms. With analytical algorithms it can be further distinguished between *closed-form* and *algebraic elimination* methods. Closed-form methods find the rotations of the joints by evaluating closed-form equations. These methods only work with very simple well posed and over-constrained problems. In case the problem is modeled using polynomial equations, which are at least in part of order higher than four or contain multiple variables, no closed-form solution exists. Then algebraic elimination is used. Despite the fact that the roots of high-order polynomials have to be found numerically, methods based on algebraic elimination are still considered analytic.

The advantages of analytical methods is their computational efficiency and that they are relatively robust in the vicinity of singularities, which cannot be guaranteed for numerical algorithms.

Numerical methods use an initial guess for the joint angles to iteratively approximate the solution. [TGB00] distinguishes between three basic approaches of numerical algorithms.

- The first approach models the problem as a system of nonlinear equations:

  $f(v) - G = 0$ where $f(v)$ again is the forward kinematics mapping function and rotation and position of the end-effector is given by G. The solution to the IK problem is given by the roots of the system. These are found by linear approximation employing the Newton-Raphson method. The problem with this algorithm is, that it only converges slow if at all in case the equations are highly nonlinear. In variations of this algorithm $f(v)$ is not seen as homogeneous transformation, but as e.g. a screw motion [TGB00].

- Alternatively the IK problem can be expressed as a differential equation. By integration of the joint velocities the joint angles can be calculated.

- The third group of algorithms tries to formulate a scalar potential function $P(v)$ that expresses the error between G and $f(v)$ by:

  $P(v) = (f(v) - G)^T \cdot (f(v) - G)$

  $P(v)$ is positive for all v and has a global minimum. The minimum can be found by any optimization algorithm that takes general nonlinear functions as an input (e.g. the nonlinear conjugate gradient method as described in section 3.6)

2.7 Quaternions

They can be thougth of as a 4D-vector, as a complex number with three imaginary parts or as a combination of a 3D-vector with a scalar. In this work the notation from [Hor87] is used denoting quaternions with a circle on top and their components as follows.

$\overset{\circ}{q} = q_0 + iq_x + jq_y + kq_z = \left[\overrightarrow{v}, s\right] = [(q_x, q_y, q_z), q_0]$ where $q_x, q_y, q_z, q_0, s \in \mathbb{R}$ and $\overrightarrow{v}$ is a 3D-vector

i, j and k have special properties:

$i^2 = -1, j^2 = -1, k^2 = -1$ and

$ij = k, jk = i, ki = j, ji = -k, kj = -i, ik = -j$

Using the notation $\left[\overrightarrow{v}, s\right]$ a 3D-vector can be easily expressed using a quaternion with $\left[\overrightarrow{v}, 0\right]$. Also a scalar can be written as $[(0, 0, 0), s]$.

The magnitude of a quaternion $\overset{\circ}{q}$ is $\left\|\overset{\circ}{q}\right\| = \sqrt{q_0^2 + q_x^2 + q_y^2 + q_z^2}$. If the magnitude of a quaternion equals 1 it is called a unit quaternion. These are especially useful, when it comes to describing rotations, as we will se later.

The conjugate of a quaternion is defined as $\overset{\circ}{q}^* = q_0 - iq_x - jq_y - kq_z = \left[\overrightarrow{-v}, s\right]$ and the inverse as $\overset{\circ}{q}^{-1} = \frac{\overset{\circ}{q}^*}{\left\|\overset{\circ}{q}\right\|}$

Given two quaternions p and q addition and subtraction are simply calculated by applying the corresponding operations on the elements of the quaternions, i.e. $r + q = \left[\left(\overrightarrow{v}_r + \overrightarrow{v}_q\right), s_r + s_q\right]$

Multiplication, however, is a little bit more complicated. There are two ways of viewing the products of two quaternions. The first is to use the representation of quaternions as complex numbers. Here every possible combination of elements of the two numbers is multiplied and added or subtracted depending on the properties of i, j and k.Since multiplications of these numbers are noncommutative multiplication of quaternions is also not commutative.

A more concise view is that of expressing the product of two quaternions as the multiplication of a $4x4$ matrix with a 4D-vector. Therefore one of the quaternions has to be expanded to an orthogonal $4x4$ matrix. The product of two quaternions then looks like this:

$$\overset{\circ}{r}\overset{\circ}{q} = \begin{bmatrix} r_0 & -r_x & -r_y & -r_z \\ r_x & r_0 & -r_z & r_y \\ r_y & r_z & r_0 & -r_x \\ r_z & -r_y & r_x & r_0 \end{bmatrix} \overset{\circ}{q} = R\overset{\circ}{q} \text{ or}$$

$$\overset{\circ}{q}\overset{\circ}{r} = \begin{bmatrix} r_0 & -r_x & -r_y & -r_z \\ r_x & r_0 & r_z & -r_y \\ r_y & -r_z & r_0 & r_x \\ r_z & r_y & -r_x & r_0 \end{bmatrix} \overset{\circ}{q} = \bar{R}\overset{\circ}{q}$$

It can be seen that the lower right $3x3$ matrix of the second $4x4$ matrix is the transpose of the corresponding $3x3$ matrix of the first $4x4$ matrix. This shows that the multiplication of quaternions is noncommutative.

The dot product of two quaternions is the product of the quaternions's elements.

$\overset{\circ}{p} \circ \overset{\circ}{q} = p_0q_0 + p_xq_x + p_yq_y + p_zq_z$

2.7.1 Special Properties of Quaternions

Quaternions have some special properties that will help later to derive other calculation rules and algorithms. These properties will be denoted as QP with the according number.

1. The expanded matrix of the conjugate of a quaternion is just the transpose of the quaternions matrix (i.e.$\overset{\circ}{q}^*p = Q^Tp$).

2. Since the expanded matrices of quaternions are orthogonal the product of such a matrix with its transpose is diagonal:
   $QQ^T = \overset{\circ}{q} \circ \overset{\circ}{q} I$, where I is the $4x4$ identity matrix. Also $\overset{\circ\circ*}{q\,q} = \overset{\circ}{q} \circ \overset{\circ}{q}$.

3. $(\overset{\circ\circ}{p\,q}) \circ \overset{\circ}{r} = \overset{\circ}{p} \circ (\overset{\circ\circ^*}{r\,q})$, for proof see [Hor87] .

2.7.2 Rotation using Unit Quaternions

Unit-quaternions are one comfortable way - among many others -to describe rotation. So, given a quaternion $\overset{\circ}{r}$ with purely imaginary components and a unit quaternion $\overset{\circ}{q}$ the composite product $\overset{\circ}{r}' = \overset{\circ\circ\circ*}{q\,r\,q}$ has special properties. Viewing the unit-quaternion $\overset{\circ}{q}$ in its vector-form $\left[\overrightarrow{v}, s\right]$ it can in fact be proven that $\overset{\circ}{r}'$ is the same vector that would result from rotating $\overset{\circ}{r}$ around axis $\overrightarrow{v}$ by the angle s.

The rotation described by a unit-quaternion can be easily transformed into a rotation matrix. This is done by expanding $\overset{\circ}{q}$ into its matrix representation:
$\overset{\circ}{r}' = \overset{\circ\circ\circ*}{q\,r\,q} = (\bar{Q}\overset{\circ}{r})\overset{\circ*}{q}$ due to the multiplication rules equals $\bar{Q^T}(Q\overset{\circ}{r})$ or $(\bar{Q^T}Q)\overset{\circ}{r}$. The product of $\bar{Q^T}$and Q can then be written as:

$$\bar{Q^T}Q = \begin{bmatrix} \overset{\circ\circ}{q\,q} & 0 & 0 & 0 \\ 0 & (q_0^2 + q_x^2 - q_y^2 - q_z^2) & 2(q_x q_y - q_0 q_z) & 2(q_x q_z + q_0 q_y) \\ 0 & 2(q_y q_x + q_0 q_z) & (q_0^2 - q_x^2 + q_y^2 - q_z^2) & 2(q_y q_z - q_0 q_x) \\ 0 & 2(q_x q_z - q_0 q_y) & 2(q_y q_z + q_0 q_x) & (q_0^2 - q_x^2 - q_y^2 + q_z^2) \end{bmatrix}$$

The lower-right $3x3$ matrix is then the rotation matrix R that can be used to rotate a vector r. The mathematical proof, that this relationship between quaternions and matrices is true, can be found in [Hor87]. Also the formulas for transforming a rotation matrix into a quaternion can be found there.

2.8 Matrices

This section presents some special properties and features of matrices, that will be used later to explain the algorithms used. It is not intended as an introduction to matrices. Thus, some understanding of matrices and its basic operations is recommended.

2.8.1 Eigenvalues and Eigenvectors

Given a matrix $A \in \mathbb{R}^{nxn}$. A real value λ is called the eigenvalue of A, if a vector $x \in \mathbb{R}^n, x \neq 0$ exists, with $Ax = \lambda x$. Then x is called an eigenvector of matrix A with eigenvalue λ.

2.8.2 Determinant

A determinant is defined for every matrix $A \in \mathbb{R}^{nxn}$ and denoted as $\det(A)$.

- For a $2x2$ matrix A with

$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$, the determinant is defined as $\det(A) = \begin{vmatrix} a & b \\ c & d \end{vmatrix} = ad - bc$

- For A being a nxn matrix with $n \geq 3$

$\det(A) = \sum_{j=1}^{n} (-1)^{1+j} \cdot a^{1+j} \cdot \det(A_{1j})$ where A_{1j} is the matrix that results from the deletion of the first row and jth column of A.

2.8.3 Characteristic Polynomial

The characteristic polynomial p_A of matrix $A \in \mathbb{R}^{nxn}$ is defined as $p_A(\lambda) = \det(A - \lambda I)$, where I is the identity matrix of same dimension as A. Note that the order of p_A is n.

The roots of p_A are the eigenvalues of A and the corresponding eigenvectors can be calculated by solving the linear equation system $(A - \lambda I)x = 0$.

Chapter 3

Design

This chapter points out the design considerations for our skeleton parameterization system. The first part of the following presents an analysis of the intended purpose and goals of the system. Secondly an overview of the tasks necessary to obtain a parameterized skeleton from optical marker data is given. In the sections thereafter I describe in more detail the parts of the system as well as algorithms used to implement the aforementioned tasks.

3.1 Overview

Figure 3.1 shows the tasks and their input/output data as it is passed through the different stages. The tracker server is only shown as black box here. Please refer to [Meh06] for more details on the tracking process of our system. The design otherwise follows the general approach described in [KOF05], although some modifications where made, as is described later.

3.2 The Goals of the System

As is pointed out in the chapters 1 and 2 a multitude of motion capture systems exists for a variety of application areas. None of the systems, however, no matter how sophisticated they are, can be used universally. Therefore it is important to isolate the main purpose so the design can be created accordingly.

The motion capture system, being developed during this thesis, is intended primarily for use in a research and teaching environment. On the one hand it is intended to provide a testbed for staff members to try out new algorithms and methods. On the other hand students should be able to use it in their projects and course work assignments (e.g. creating animations for small computer games).

The main design goals are therefore flexibility and a straightforward handling, which enables the students to use the system with only very little instructions. The first is achieved by using a modular structure with clearly specified interfaces.

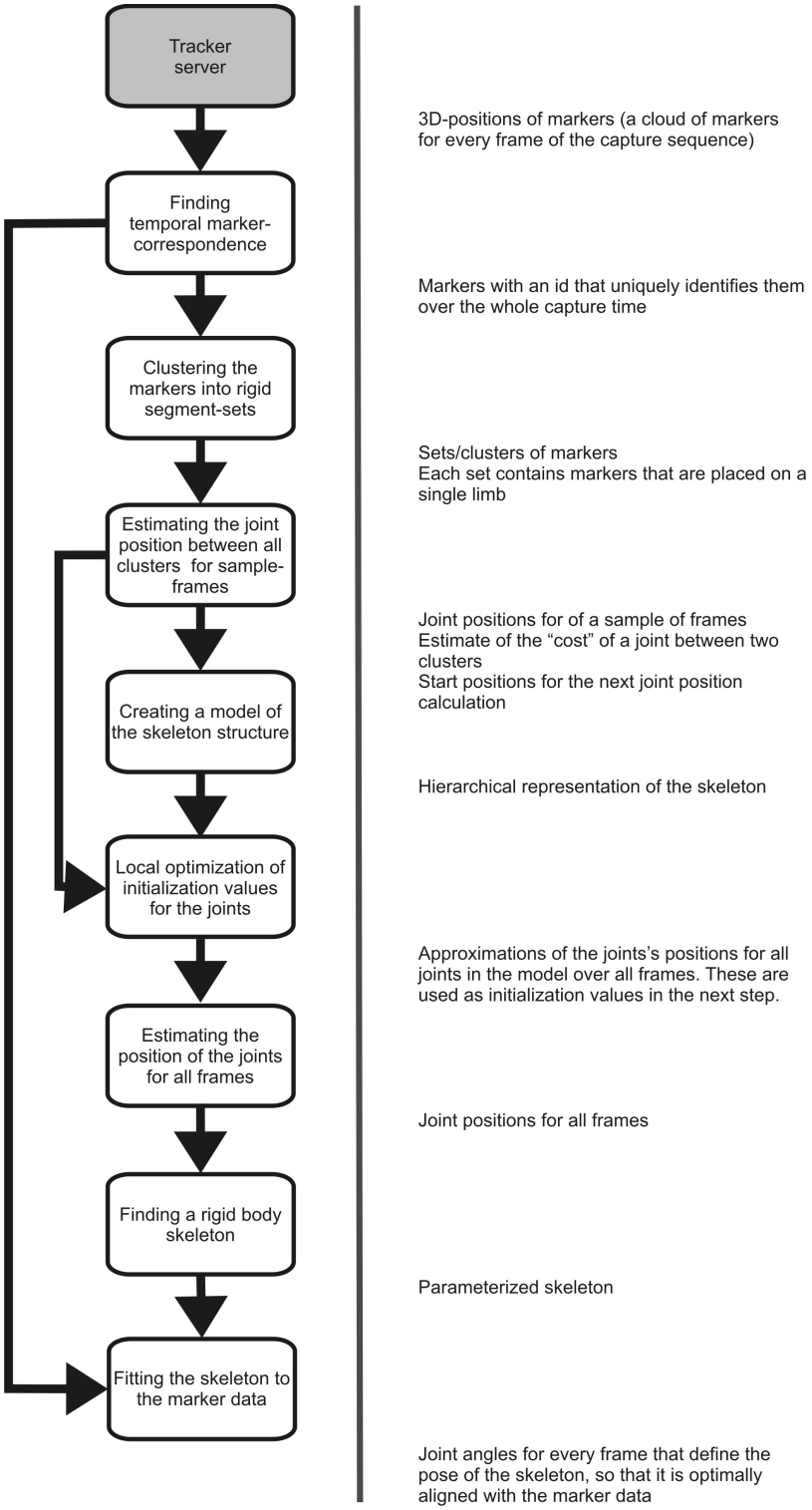

Figure 3.1: Overview of the tasks and their input/output

The latter is ensured by making the algorithms perform the operations automatically, reducing the necessary user interaction to a minimum. Furthermore the system adapts to different users without any additional effort, like measurements of the limbs. In addition there is no limitation to human motion capture; virtually any articulated structure can be captured, as long as it has a certain size and the articulations can exercise the necessary degrees of freedom. This includes for example puppets or animals (as long as they are convinced to stay in the capture area and perform gym motions) These features, however, come with a price, as presented in section 6.2.

The following sections describe the tasks depicted in 3.1 in more detail and discuss the employed algorithms. Firstly section 3.3 explains, how a marker is identified over all frames. Then section 3.4 describes how the markers that are placed on the same limb can be grouped/clustered together. After that section 3.5 presents a method to estimate the joint positions and refine their precision. Finally, in section 3.8 the parameters of the skeleton model are calculated and the pose of parameterized skeleton is inferred from the marker data.

3.3 Temporal Marker Correspondence

The input to the first stage comes from the tracker server. It consists of the three-dimensional positions of the markers as they are recorded at discrete times during the capture interval. Since we are capturing about 40 markers at a rate of 200Hz this gathers to be a substantial amount of data. As pointed out in [Meh06] a passive optical tracking system is used, which has the disadvantage of not relaying identities with the markers. Thus for every frame a list of positions exits with no information whatsoever which marker they belong to. This brings us to the first task to be performed, which is finding the physical markers that the clouds of 3D points, stored for every frame, belong to. In other words, the temporal correspondence between the points has to be found. For this matter numerous algorithms have been developed throughout the years. Mostly only two consecutive frames are used and the points of the later frame are assigned to those of the earlier. Some systems, however, employ more complex techniques using for example marker trajectories trying to predict the marker position in the next frame. Some of these algorithms use methods applied on the camera's image plane, while others solely are performed on 3D coordinates.

[GF05] for example suggests an image space algorithm that employs intensity and distance measurements to determine which point of two succeeding frames are to be matched. For every possible match of points of the two succeeding frames the difference of the intensity values, obtained by the cameras, and the distance is measured. From these measurements a value is calculated, that determines the "strength" by which the two points belong to the same marker. The values are fed into a strength matrix which then is used in turn to determine the optimal matches.

[RL02] on the other hand uses a technique called Lagrangian relaxation to estimate the association of 2D positions in the camera‘s image plane with the markers. With this method both spatial and temporal correspondence are obtained. The drawback of Lagrangian relaxation, however, is that it is not guaranteed to come up with the optimal solution, but delivers an estimate of how close it is to the optimum.

In our system we are basically using the method as described in [KOF05]. It operates solely on the 3D points generated by the tracker from the camera's captured image data. Relying on the fact that markers usually move very little between successive frames, identities are traced from frame to frame. Since the tracker captures at a relatively high rate and is not intended to track people performing very fast motions (e.g. sports or fighting animation) this method can be expected to work well. As long as no marker disappears due to occlusion it provides a fast and simple solution to the association problem. How the algorithm responds to lost markers is described later on in this section. The following steps are executed to obtain the temporal correspondence of the markers:

1. The 3D positions of the first frame are labeled with unique identities.
2. The points of the second frame are matched with the points of the first frame. Therefore two sets of points are created for which a one-to-one mapping has to be found. For this reason the Euclidean distance between all points of the two sets is calculated.
3. Then the two points with the smallest distance are marked as a match and removed from the sets.
4. Repeat 3. until the two sets are empty.
5. For all matches assign the point of the second frame the id of the point of the first frame.
6. Make the second frame the first frame. If the second frame isn‘t the last frame, make the next frame the second and go to 2.

As mentioned before, this method works very well for perfect data and for the tests with artificial data. Note that this has only been tried for test purposes. To prevent unnecessary errors in the test data the ids of the markers are retrieved from the csm file. See section 4.12 for details on the csm file format. In practice, however, markers get lost and reappear during the tracking process, due to temporal occlusion by clothing or limbs. Even worse the precision of the marker‘s 3D position-reconstruction might seriously suffer shortly before they disappear/ after they reappear. This results in markers “jumping around”, thus changing their position dramatically in consecutive frames. Since for our tracking system that doesn‘t occur too often, these problematic markers are simply discarded. Therefore a threshold, which marks the maximum a marker is allowed to move

from one frame to another, is used. Markers that "jump" over this threshold are then ignored for a short period of time. The threshold as well as the time delay, which shouldn't be more than a couple of frames, depend on the tracking system used. To retrieve the identity of a reappeared marker, however, a more sophisticated algorithm has to be applied.

A marker that reappears, after it has been occluded, is treated as if it were a new marker in the first frame and assigned a new identity. Thus, one physical marker on the actor‘s body suit can have multiple identities over the captured time span. As long as that only happens to very few markers it could very well be ignored and every identity treated as if it were a marker on the tracked subject. For more markers disappearing/reappearing, however, this results in a degradation of the following stages‘s accuracy. Therefore an algorithm is needed to find all those identities, that belong to one marker, and merge them into a single data set.

For this purpose [KOF05] suggests a method that exploits the fact, that the tracked subject often moves through the same poses. When two frames are identified to show the same pose and two markers have similar coordinates, it is very likely that they correspond to the same physical marker. The pose, however, is subjected to global rotation and translation, which has to be removed before the positional data of markers can be compared.

To accomplish this task [KOF05] introduces a data structure they call *Marker Set*. Each of the n virtual markers, for which an identity is found, has its own *Marker Set*. A *Marker Set* contains the data of all the frames for which the virtual marker exists and thus holds all information available about it as well as the relationships to other markers. I will use a nomenclature consistent with [KOF05] in this section, thus naming the *Marker Set* of the ith marker p_i. These n *Marker Sets* are now to be grouped together into the n' physical markers. n' is assumed as the maximum number of markers captured in a single frame. So, if no frame exists, that contains all markers, the algorithm will evidently fail to deliver a correct result.

To determine which *Marker Sets* should be grouped and which shouldn't a distance function is created. This distance function allows us later to use a clustering algorithm on the *Marker Sets.* The distance D_{ij} between a *Marker Set* p_i and a *Marker Set* p_j according to [KOF05] is defined as the minimum distance between markers i and j over all pairs of poses, that is

$$D_{ij} = \min_{a \in p_i, b \in p_j} \|m_{i,a} - Am_{j,b}\|$$

Here a and b denote single frames of the *Marker Sets* p_i and p_j. $m_{i,a}$is the position of marker i in frame a, while A is the matrix that performs the global rotation and translation necessary to align the pose in frame b with that in frame a. So, for all frames of p_i, the distance to all frames of p_j is calculated one frame at each side at a time. Then the minimum of those distances is considered the distance between the *Marker Sets.*

To be able to calculate these distance values one assumption has to be made.

It is crucial that the *Marker Sets* have at least three markers in common, of which the identity is already known. Otherwise A and thus the rotation and translation can not be estimated, which, however, should never happen anyway. If there are less than three markers, it would mean in our case, that the tracker has lost more than 90 percent of the markers. That in turn would mean serious trouble for the other stages as well, because they are depending on continuous data.

Since each *Marker Set* has hundreds or even thousands of frames, it is evident that calculating the distance between all pairs of frames of two sets generates a gigantic computational overhead. Due to the fact that there are no huge changes of the pose in consecutive frames, especially at high capture rates, the calculations can be speed up significantly. [KOF05] suggests using only sample frames from every *Marker Set.* Starting from the sample frames that had the smallest distance, neighboring frames can be evaluated as well. Results prove that this optimization produces the same matches as the original algorithm.

Furthermore it has to be taken into account that *Marker Sets* which overlap can not belong to the same physical marker. Thus calculating the distance of *Marker Sets* that have frames in common is avoided setting it to the highest possible value.

Finally, having calculated the distance values of all pairs of *Marker Sets,* a clustering algorithm can be applied to group the sets together, producing n' markers. For that purpose Spectral Clustering is used, which is explained in more detail in 3.4.2.

3.4 Clustering Markers

3.4.1 Concept

After the 3D- trajectories of all markers have been found, the next step is to determine which markers are placed on the same limb and group them together. This can be done either manually or automatically. In the manual case the user selects the marker-limb association for the first frame. Due to the fact that the temporal correspondence between the captured markers is known through the trajectories these associations apply for all captured frames.

This manual labeling task, however, requires an extra effort and user intervention, which is not welcomed in most systems. Therefore automatic methods for partitioning markers have been developed, like those suggested in [KOF05] [SPB$^+$98] [GF05]. These algorithms are all using measurements of the distance between markers to determine marker-limb associations. The basic idea is, to assume that a limb of a human body is an (almost) rigid object. Thus two markers placed on a single limb are never to move apart from or closer to each other, but always remain at a constant distance. In practice, however, the markers of one segment will move relative to each other due to skin or muscle movements, an ill-fitting body suit, as described in 4.14, or measurement errors of the tracking system. So the algorithm has to determine whether the movement comes from

errors or - in case it is large enough- from the bend or twist motion of a joint. As a measurement of the distances' alteration, the variance of the distance over all frames is used. Therefore first the average distance $avgDist_{ij}$ between all markers i and j, where $i \neq j$, has to be calculated.

$$avgDist_{ij} = \frac{1}{n_f} \times \sum_{n_f} \sqrt{(mx_i - mx_j)^2 + (my_i - my_j)^2 + (mz_i - mz_j)^2}$$

Here n_f is the number of all frames and mx_i, my_i and mz_i are the coordinates of the marker i respectively. The variance of the distance $varDist_{ij}$ between i and j can then be determined by

$$varDist_{ij} = \frac{1}{n_f} \times \sum_{n_f} (\sqrt{(mx_i - mx_j)^2 + (my_i - my_j)^2 + (mz_i - mz_j)^2} - avgDist_{ij})^2$$

Using these variances a cost matrix W is created, where the elements W_{ij} are depending on the variance $varDist_{ij}$. W is symmetric and elements in the major diagonal are zero.

Section 3.4.2 explains how the cost matrix is used by a Clustering algorithm, namely Spectral Clustering, to calculate a marker-limb association. The number of clusters (the number of sections/limbs of the articulated model) that are to be created has to be specified by the user. In a different approach the number of clusters could be inferred automatically by clustering multiple times with a different number of clusters. The number of clusters is increased until the maximum variances of distance within one cluster are below a certain threshold. This value, however, is difficult to determine and changes with tracking precision and marker configuration. [ZMP04] suggests a method that evaluates the eigenvectors, created during the clustering process, to determine the number of clusters. This, however, is computationally not trivial and additionally increases the risk of creating a wrong cluster.

For example the torso section is a difficult matter, since the model generated by our method is supposed to have only two clusters - one for the hip and one for the breast/back. Dividing that section into more clusters might result in problems with the joint estimation (e.g. only two markers for two adjacent clusters would make it impossible to make a useful prediction of the joint between them). We, therefore, have the user specify the number of limbs the tracked subject is supposed to have.

Besides Spectral Clustering other algorithms, using variance of distance, have been proposed, e.g. in [RL02] [SPB+98]. These for example are using a threshold to group the markers together. All marker-pairs that have an entry in the cost matrix below that threshold are considered being on the same segment. The threshold is set to a value so that exactly the user-specified number of clusters are being produced. This straight-forward approach is decisively faster than Spectral Clustering from a computational point of view. Compared to the process of finding joint positions as described in 3.5 , however, the expense is relatively

small. Therefore, in our system the Spectral Clustering algorithm is favored over the fast approach, which might not perform very well under harsh conditions.

As was pointed out in 3.3 poses and thus distances between markers placed on different limbs do not change much in consecutive frames. Therefore, sampling can be used again to reduce the number of frames that have to be considered and thus to speed up the calculations. [KOF05] suggests to take samples every half second with a jitter of some thirtieths of a second. The jitter ensures that periodic errors only have little influence on the outcome of the clustering. Sampling changes the number of frames n_f in the formulas above to n_{sf}, the number of sampled frames, but otherwise leaves the algorithm intact producing correct clusters. For good data even as little as ten frames are enough for the clustering, as is shown in 5.

To ensure optimal clustering and remove the effect of sporadic jumps of markers the system described in [KOF05] samples and clusters multiple times. The solution that has the smallest standard deviation of distance within its clusters is chosen to be the optimal. For our system this variation does not seem to be necessary, because jumping markers are already filtered at an earlier time. So markers that move more than a certain threshold from one frame to another are simply discarded as is described in 3.3.

3.4.2 Spectral Clustering

According to [JMF99] "clustering is the unsupervised classification of patterns (observations, data items, or feature vectors) into groups (clusters)". Clustering is used in a variety of disciplines, like image segmentation, information retrieval and pattern/ object recognition/ classification. What is considered a good clustering in one of these fields, however, is not necessarily one in another application. Therefore lots of clustering algorithms exist for different application areas.

A class of methods, which has been developed and improved in the last couple of years, is Spectral Clustering. These methods rely on the spectral analysis of a distance(or affinity) matrix .Distance here has to be understood not only as the Euclidean distance, but is used in a broader sense as a measurement of similarity between two objects. As mentioned before, this matrix in our case is created using the variance of distance between markers over time. To analyze the matrix, concepts from spectral graph theory are used. Graph theory derives properties of a graph from the graphs affinity matrix or more precise from the Laplacian of the matrix. The main idea of Spectral Clustering, thus, is to first transform the clustering problem into a graph partitioning problem using the data objects as nodes and the affinities as edge weights. Then the spectral graph theory is used to change the representation of the objects, that ought to be clustered, so cluster-properties are amplified. After that the alternate representation is clustered using some clustering algorithm (usually k-means is used). The results of the clustering algorithm finally are re-converted into the original form of the data.

Spectral Clustering has many advantages over other clustering algorithms. So

it is easy to implement, because many linear algebra packages and libraries exist, that aid the programmer. Furthermore Spectral Clustering delivers better results than most traditional algorithms on non-trivial clustering problems. Additionally it doesn't require restarts with different parameterization and can't get trapped in local minima. Parameterization of the affinity matrix, although, might be an issue, as will be pointed out later in this chapter.

Nevertheless, it is not easy to see at first glance why Spectral Clustering actually works. Therefore, three approaches have been developed to explain the algorithm: graph cut (e.g. in [YS03]), random walk (e.g. in [MS00]), and perturbation theory (e.g. in [ZMP04] and [NJW02]). These different points of view are summed up in [vL06]. I consider the graph cut approach the most straight-forward and therefore used it in this thesis to derive Spectral Clustering. To keep things simple, however, some mathematical proofs are left aside and can be looked up in [vL06] or [YS03].

The next two sections present the graph notation used and the definition and properties of the Laplacian. Then follows the graph cut approach and finally the clustering algorithm itself is presented.

Graph Notation and the Similarity Graph

Before spectral graph theory can be used for clustering the data objects/points $S = \{s_1, s_2, ..., s_n\}$ have to be mapped on a so-called similarity graph. Different graphs exist for that matter, like $\varepsilon-$neighborhood graph, k-nearest neighbor graph or fully connected graph. The main purpose of such a graph is to represent local neighborhood relationships. For our system a fully connected graph is used. Here all vertices v_i and v_j are connected and weighed with the value w_{ij} as it is produced by the similarity function. The graph used is represented by $G = (V, W)$ with the set of vertices $V = \{v_1, v_2, ...v_n\}$ and the adjacency matrix W. The adjacency matrix contains the edge weight between all vertices. Weight between two vertices v_i and v_j is denoted with w_{ij} and given by the Gaussian similarity function $w_{ij} = e^{\frac{-distance^2(s_i,s_j)}{2\sigma^2}}$.Furthermore $w_{ij} = w_{ji}$ for all $i, j = 1...n$ making G an undirected graph. The degree of a vertex v_i is given by $d_i = \sum_{j=1}^{n} w_{ij}$,which is the sum of W's ith row. The degree matrix D is a diagonal matrix with values $d_1, d_2, ...d_n$ on the diagonal.

A partition is a set $\{A_1, A_2....A_k\}$ of subsets of V, where $A_i \cap A_j = \varnothing$ for $i \neq j$ and $A_1 \cup A_2... \cup A_k = V$. For simplification the set of indices $\{i|v_i \in A_j\}$ is denoted as $i \in A$. The complement $V \backslash A_i$ of a subset $A_i \subset V$ is denoted as $\bar{A}_i$. The number of vertices contained in A_i is $|A_i|$. Another measure for the size of A_i incorporating the weights of its edges is given by $vol(A_i) = \sum_{i \in A_i} d_i$.

The transpose of a vector f or a Matrix M is denoted by f^T and M^T. Furthermore the constant one vector $(1, 1, ...1)$ is specified by $\imath$: Similarly the vector $\imath_{A_i}$, has 1 on the ith place, if $v_i \in A_i$ and 0 otherwise.

The Laplacian

As mentioned before Spectral Clustering uses the Laplacian matrix of a graph for its analysis. In the literature multiple definitions of the Laplacian exist and therefore one has to be careful of which to use and how they are denoted (see [vL06] for a discussion on that matter). Starting with a graph as described in the last section the unnormalized Laplacian L is defined as

$$L = D - W$$

The properties of L, which are relevant for Spectral Clustering, are then

1. For all vectors $f \in \mathbb{R}^n$
$$f^T L f = f^T D f - f^T W f = \sum_{i=1}^{n} d_i f_i^2 - \sum_{i,j=1}^{n} w_{ij} f_i f_j =$$
$$= \frac{1}{2} \left(\sum_{i=1}^{n} d_i f_i^2 - 2 \sum_{i,j=1}^{n} w_{ij} f_i f_j + \sum_{j=1}^{n} d_j f_j^2 \right) =$$
$$= \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(f_i - f_j \right)^2$$

2. L is symmetric and positive semi-definite (Since W is symmetric and D is a diagonal matrix , L has to be symmetric too. $f^T L f \geq 0$ characterizes a positive semi-definite matrix, which easily proofs true, when looking at 1.)

3. L has the constant one vector $\imath$ as eigenvector, which has 0 as eigenvalue. (can be easily seen in 1.)

4. L has n non-negative real eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$ (Due to 2. and 3. L is a Hermitian matrix, which is why the finite-dimensional spectral theorem can be applied resulting in property 4)

The normalized Laplacian L_{sym} is here defined as

$$L_{sym} = D^{-1/2} L D^{-1/2}$$

Note that in the algorithm later we will apply $L_{sym2} = I - L_{sym}$ instead (i.e. $L_{sym2} = D^{-1/2} W D^{-1/2}$). For the analysis of Spectral Clustering in the next sections L_{sym} is examined, because it is easier to handle. The subtraction from the identity matrix, however, doesn‘t change much for the algorithm. Only the eigenvalues λ of the matrix change to $1 - \lambda$. For more details on this matter please refer to [KOF05]

The Laplacian as defined above has the following specific properties that will be used in the next sections to derive Spectral Clustering.

1. For all vectors $f \in \mathbb{R}^n$

   $f^T L_{sym} f = \frac{1}{2} \sum_{i,j=1}^{n} w_{ij} \left(\frac{f_i}{\sqrt{d_i}} - \frac{f_j}{\sqrt{d_j}} \right)^2$

2. 0 is an eigenvalue of L_{sym} with eigenvector $D^{1/2}\imath$.

3. L_{sym} is positive semi-definite

4. L_{sym} has n positive real eigenvalues $0 = \lambda_1 \leq \lambda_2 \leq ... \leq \lambda_n$

5. L_{sym} as L is a Hermitian matrix.

The properties of the normalized Laplacian can be proven similarly to those of the unnormalized one, as can be seen in [vL06].

The Graph Cut Approach

This section shows how Spectral Clustering can be derived as an approximation to graph partitioning problems. When a coherent graph is split up into multiple partitions, two objectives have to be considered. First of all edges between two different groups(those that are "cut") are supposed to have low weight. Secondly, edges within a partition should have high weight. Given a graph $G = (V, W)$ for these objectives the following measurement functions can be defined.

For A and B being two subsets of V and $A \cap B = \varnothing$ we define $cut(A, B) = \sum_{i \in A, j \in B} w_{ij}$.

The task of finding the partition $\{A_1, A_2....A_k\}$ that has minimal edge weight between groups is called mincut problem. It can be written as

$$\text{minimize } cut(A_1, A_2....A_k) = \sum_{i=1}^{k} cut(A_i, \bar{A}_i)$$

The best solution to the mincut problem, as it is written here, however, would be to separate single vertices from the graph. This is usually not desirable, because the groups are expected to have "reasonable" size. Therefore the objective function has been extended. In [vL06] two functions are suggested: RatioCut and Ncut. Here Ncut is used, which is defined as

$$Ncut(A_1, A_2....A_k) = \sum_{i=1}^{k} \frac{cut(A_i, \bar{A}_i)}{vol(A_i)}. \qquad \text{where} \left(vol(A_i) = \sum_{i \in A_i} d_i \right)$$

When Ncut is minimized the division by the sum of edgeweights, that exert from a partition, ensures that larger groups don‘t have a disadvantage over small groups. The mincut problem, however, formulated in this way, is NP hard. In the following it is shown how Spectral Clustering is derived as a relaxation of this problem, which makes use of the Rayleight-Ritz theorem.

The Rayleigh-Ritz Theorem The theorem works with Hermitian matrices, which can be complex as well as real. In our case the matrices contain only real numbers and are symmetric, which makes them a special case of Hermitian matrices. Only these are considered here.

According to [Mey00] given a Hermitian matrix $A \in \mathbb{R}^{nxn}$ A's eigenvectors are the critical points (vectors) of the "Rayleigh quotient". The eigenvalues of these vectors are the values at the critical points. The quotient is a real function $R : \mathbb{R}^n \longrightarrow \mathbb{R}$ with

$R(x) = \frac{x^T Ax}{x^T x}$ with $\|x\| \neq 0$

Consequentially the minima of this function are determined by the eigenvectors with the smallest eigenvalues. A minimization problem, which is brought into the form of the theorem, thus, can easily be solved using eigendecomposition.

Approximation of Ncut This sub-section presents a way to reformulate the Ncut-problem using the Laplacian so the Rayleigh-Ritz theorem can be used to approximate it. At first it is shown, how Spectral Clustering works for $k = 2$ partitions. Vertices can, therefore, only be in partition A or in partition $\bar{A}$ The goal therefore is to optimize the problem

$\min_{A \subset V} Ncut(A, \bar{A})$.

The first step is to find an indicator vector f, so that $\min_{A \subset V} c * Ncut(A, \bar{A}) = \min_{A \subset V} f^T L f$ (c being some constant)

For this reason f is defined as

$$f_i = \begin{cases} \sqrt{\frac{vol(\bar{A})}{vol(A)}} & \text{if } i \in A \\ -\sqrt{\frac{vol(A)}{vol(\bar{A})}} & \text{if } i \in \bar{A} \end{cases}$$

Using the unnormalized Laplacian the $Ncut$ function can be derived

$$f^T L_{sym} f = \sum_{i,j=1}^{n} w_{ij} (f_i - f_j)^2 =$$

$$= \sum_{i \in A, j \in \bar{A}} w_{ij} \left(\sqrt{\frac{vol(\bar{A})}{vol(A)}} + \sqrt{\frac{vol(A)}{vol(\bar{A})}} \right)^2 + \sum_{j \in A, i \in \bar{A}} w_{ij} \left(-\sqrt{\frac{vol(\bar{A})}{vol(A)}} - \sqrt{\frac{vol(A)}{vol(\bar{A})}} \right)^2 =$$

since $cut(A, \bar{A}) = \sum_{i \in A, j \in \bar{A}} w_{ij}$ we can combine the sums (of the now calculated squares)

$2cut(A,\bar{A})\left(\frac{vol(\bar{A})}{vol(A)}+\frac{vol(A)}{vol(\bar{A})}+2\right)=2cut(A,\bar{A})\left(\frac{vol(\bar{A})+vol(A)}{vol(A)}+\frac{vol(A)+vol(\bar{A})}{vol(\bar{A})}\right)=$

$2vol(V)cut(A,\bar{A})\left(\frac{1}{vol(A)}+\frac{1}{vol(\bar{A})}\right)=2vol(V)Ncut(A,\bar{A})$

In Addition we can proof, that Df is orthogonal to the constant one vector $\imath$ by

$(Df)^T\imath=\sum_{i=1}^{n}d_if_i=\sum_{i\in A}d_i\left(\sqrt{\frac{vol(\bar{A})}{vol(A)}}\right)+\sum_{i\in\bar{A}}d_i\left(-\sqrt{\frac{vol(A)}{vol(\bar{A})}}\right)=$

which, due to the definition of $vol()$ can be reduced to

$vol(A)\left(\sqrt{\frac{vol(\bar{A})}{vol(A)}}\right)-vol(\bar{A})\left(\sqrt{\frac{vol(A)}{vol(\bar{A})}}\right)=\sqrt{vol(A)}\sqrt{vol(\bar{A})}-\sqrt{vol(\bar{A})}\sqrt{vol(A)}=0$

Thus $Df\perp\imath$ is true. Similarly $f^TDf=vol(V)$ can be proven.

By plugging things together, the minimization of $Ncut$ can be written down as

$\min_A f^TLf$ where $Df\perp\imath$ and $f^TDf=vol(V)$

This is a discrete optimization problem, since the entries of the solution vector f can only have the two particular values. The minimization problem then is relaxed by ignoring the condition of discrete values for allowing $f\in\mathbb{R}^n$. This, although making an exact solution impossible, results in a problem that is much easier to solve.

$\min f^TLf$ where $Df\perp\imath$ and $f^TDf=vol(V)$

Then $g=D^{\frac{1}{2}}f$ is created, where $g\in\mathbb{R}^n$,and f is substituted. The orthogonality-condition and the property of $g^Tg=vol(V)$ can be determined from the conditions of f. The problem then can be rewritten as

$\min g^TD^{-\frac{1}{2}}LD^{-\frac{1}{2}}g$ where $g\perp D^{\frac{1}{2}}\imath$ and $g^Tg=f^TDf=vol(V)$

So, having brought the problem into the form of the Rayleight-Ritz theorem it can be solved with the eigenvectors of L_{sym}.Since $D^{-\frac{1}{2}}LD^{-\frac{1}{2}}=L_{sym}$, we know from 3.4.2 that $D^{\frac{1}{2}}\imath$ is the first eigenvector of L_{sym}.So the vector g, which is the eigenvector with the second smallest eigenvalue is the solution to the problem. To approximate the minimum of $Ncut$ the solution vector g, which has real values, serves as an indicator for the discrete problem of partitioning the graph. In case of $k=2$ this is done simply by assigning v_i to A if $g_i\geq 0$ and $\bar{A}$ elsewise.

Approximation of Ncut for k≥2 The approximation of $Ncut$ for $k \geq 2$ can be similarly derived as in the case of $k = 2$. At first, indicator vectors have to be defined for every partition. These k vectors are denoted $h_i = (h_{1,i}, h_{2,i}, ...h_{n,i})$ with

$$h_{i,j} = \begin{cases} \frac{1}{\sqrt{vol(A_i)}} & \text{if } i \in A_j \\ 0 & \text{otherwise} \end{cases}$$

Then the matrix $H \in \mathbb{R}^{nxk}$ is defined as the matrix, that contains the vectors h_i as columns. Since the vertices are assigned exclusively to one partition h_i is orthonormal to h_j for $i \neq j$ (all h vectors having unit length). Therefore $H^T H = I$. Furthermore $H^T LH$ is a square matrix $\in \mathbb{R}^{kxk}$ and $h_i^T Lh_i = (H^T LH)_{ii}$, where ii indexes the diagonal elements. This can be easily checked considering the rules for matrix multiplication.

Similarly to the last sub-section it can be shown that

$$h_i^T Lh_i = 2cut(A_i, \bar{A}_i)/vol(A_i)$$

Instead of h_i, h_l is written in the following so the indices can‘t be confused

$$h_l^T Lh_l = \sum_{i,j=1}^{n} w_{ij} \left(h_{i,l} - h_{j,l}\right)^2 =$$

$$= \sum_{i \in A_i, j \in \bar{A}_i} w_{ij} \left(\frac{1}{\sqrt{vol(A_i)}}\right)^2 + \sum_{j \in A_i, i \in \bar{A}_i} w_{ij} \left(-\frac{1}{\sqrt{vol(A_i)}}\right)^2 =$$

$$= 2cut(A_i, \bar{A}_i)/vol(A_i)$$

For the next step of the reformulation we are using a construction from linear algebra called a trace. The trace of a square matrix is the sum of its diagonal elements. It is denoted here as Tr. The trace of a matrix equals the sum of the matrix‘s eigenvalues. So $Ncut$ is reformulated as

$$Ncut(A_1, A_2....A_k) = \frac{1}{2}\sum_{i=1}^{k} h_i^T Lh_i = \frac{1}{2}\sum_{i=1}^{k} (H^T LH)_{ii} = \frac{1}{2}Tr(H^T LH)$$

To solve trace minimization problems a special version of the Rayleigh-Ritz theorem can be used. According to the theorem the matrix H that contains the eigenvectors with the smallest eigenvalues is the optimal solution to the problem.

$\min_{A_1,A_2....A_k} Tr(H^T LH)$ subject to $H^T DH = I$ (one can check the definition of H for proof on the latter)

Again the discreteness condition is relaxed allowing the matrix H to contain arbitrary real values. Then a matrix $U = D^{\frac{1}{2}}H$ is created substituting H, which leads to

$$\min_{A_1,A_2....A_k} Tr(U^T D^{-\frac{1}{2}} L D^{-\frac{1}{2}} U) \text{ where } U^T U = H^T D^{\frac{1}{2}T} D^{\frac{1}{2}} H = H^T DH = I$$

After calculating U, discrete partitioning information has to be retrieved from U's real-valued entries. The heuristic used for $k = 2$ cannot be used any more, since it only partitions into two clusters. Thus another method has to be found. Considering the definition of H, we know that a value in the ith row of the jth column vector of U should be relatively high if v_i in A_i and low if not. Viewing the rows of U as points in multidimensional space vertices belonging to one A_i can then be observed to have a greater distance to the other points on the ith coordinate axis. Therefore a clustering algorithm can be used to find corresponding points/rows. As is pointed out in the next section k-means clustering is used to perform this task.

To use a full-grown clustering algorithm within another clustering algorithm doesn't seem to make much sense. For many clustering problems, however, especially when clusters are not convex or don't have Gaussian distribution, Spectral Clustering performs much better than traditional algorithms. This can be seen in the experiments conducted in [NJW02].

The Spectral Clustering Algorithm

This section sums up the tasks that have to be performed for Spectral Clustering. Our system basically implements the algorithm described in [NJW02] and extends it with the local scale of analysis suggested in [ZMP04]. The notation here, though, is slightly different, so it matches with the last sections. Changes according to the extension are marked by brackets like these: "⟨⟩". The goal is to cluster a set of n objects/points $S = \{s_1, s_2, ..., s_n\}$ in $\mathbb{R}^l$ into k clusters. To achieve that, the following steps have to be taken:

1. Calculate the affinity matrix $W \in \mathbb{R}^{n \times n}$ with the elements $w_{ij} = e^{\frac{-distance^2(s_i,s_j)}{2\sigma^2}}$ ⟨$w_{ij} = e^{\frac{-distance^2(s_i,s_j)}{\sigma_i \sigma_j}}$⟩ for $i \neq j$ and $w_{ii} = 0$. $distance(s_i, s_j)$ denotes some distance function between s_i and s_j. σ is a global scaling parameter and has to be either specified by the user or determined by clustering multiple times and selecting the value which delivers the best result.

σ_i, however, is a local parameter, which describes the scale factor of s_i's neighborhood. Distances are thus scaled differently, depending on the point they have been observed from. $distance(s_i, s_j)$ is therefore scaled by σ_i to $\frac{distance(s_i,s_j)}{\sigma_i}$, when in the context of s_i, and to $\frac{distance(s_j,s_i)}{\sigma_j}$, in the context of s_j. Squaring the distance then results in $\frac{distance(s_i,s_j)distance(s_j,s_i)}{\sigma_i\sigma_j}$, which finally produces the formula $\langle W_{ij} = e^{\frac{-distance^2(s_i,s_j)}{\sigma_i\sigma_j}} \rangle$ given above.

The scale factor is dependent on the local statistics of s_i's neighbors making $\sigma_i = distance(s_i, s_m)$ as used in [ZMP04] a simple solution. The points are being sorted by distance to s_i. Here s_m is the mth point in the neighborhood of s_i. In [ZMP04] $m = 7$ is used, which seems to work well for their applications. In our case, however, that is probably not the optimal choice. Since we have substantially less points to cluster we have to choose a lower number for m.

2. Calculate, L_{sym2}, the normalized affinity matrix of W. First create the diagonal degree matrix D with the elements $D_{ii} = \sum_{j=1}^{n} w_{ij}$, which is the sum of W s ith rows, and $D_{ij} = 0$ for $i \neq j$ (of course, for D being a diagonal matrix). Use D and W to calculate the normalized affinity matrix $L_{sym2} = D^{-\frac{1}{2}} W D^{-\frac{1}{2}}$, e.g.:

$$L = \begin{bmatrix} \frac{1}{\sqrt{D_{11}}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{D_{22}}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{D_{33}}} \end{bmatrix} W \begin{bmatrix} \frac{1}{\sqrt{D_{11}}} & 0 & 0 \\ 0 & \frac{1}{\sqrt{D_{22}}} & 0 \\ 0 & 0 & \frac{1}{\sqrt{D_{33}}} \end{bmatrix}$$

3. Perform eigenvalue decomposition on matrix L_{sym2} and calculate the eigenvectors $x_{1,}x_{2,}..., x_k$ for the k largest positive eigenvalues (k being the number of clusters). For this purpose Jacobi decomposition is used, which provides a relatively slow but reliable iterative method to find eigenvectors- and values. Use these eigenvectors to form the Matrix $X = [x_{1,}x_{2,}..., x_k] \in \mathbb{R}^{n\times k}$ by making the eigenvectors columns of X.

4. Normalize the rows of X to obtain matrix Y, where $Y_{ij} = \frac{X_{ij}}{\sqrt{\sum_{j=1}^{k} X_{ij}^2}}$

   This normalization is necessary if some of the vertices in the similarity graph have particularly low degree compared to the other vertices (i.e. the edges going from a vertex of one partition to other partitions have very low weight.). In that case the entries in the eigenvectors are very small and thus for such a vector have to be scaled up. For a formal discussion of that matter please refer to [vL06].

5. Treat each normalized row of Y as a point in $k-$dimensional space $\mathbb{R}^k$ and cluster these points to form k clusters. Here we are using k-means

clustering as suggested in [NJW02], but any other clustering algorithm, that is capable of minimizing a formal objective function, can be used as well. The objective function of k-means is the means squared distance between the data points and so-called *centers*, which are points in $\mathbb{R}^k$ representing the middle of a cluster. The most popular heuristic method, that performs k-means clustering is the generalized Lloyd's algorithm. This algorithm, however, gets stuck in local minima easily, which is why our system uses a hybrid approach. Adding a heuristic that, based on local search, swaps in and out centers of the existing solution, helps escaping local minima.

6. Evaluate the result of Lloyd's algorithm. If the ith row of Y is assigned to cluster c, then s_i is to be assigned to cluster c.

For a more detailed theoretical analysis and examples of clusterings done by Spectral Clustering please refer to [vL06] and [NJW02]. In Addition [ZMP04] holds further improvements to the algorithm. [KMN+02b] and [KMN+02a] offer a more detailed description as well as an implementation of k-means clustering.

3.5 Estimating the Topology and Joint Positions

After having clustered the markers into groups for every segment of our skeleton model, the topology and the joint positions can be estimated. These two tasks are very closely related; in fact we are using the same method to infer them. To decide whether or not a joint is placed between two segments and, in case it is, to determine where to site it, a cost function is needed. Two assumptions concerning our model are necessary for this cost function to work. First of all human joints in our model are approximated by ball joints. This of course is a harsh idealization, especially for joints like the knee, where the center is moving about between one and two centimeters. For our purpose, however, this approximation is close enough. The second assumption, which has to be made, is that markers stay at the same place relative to the bones. This, however, might not be entirely true either, - skin movement and muscle deformation introduce variations - but we expect the overall error to be negligible.

Consider the case of only two segments A and B for which the assumptions apply. They are thus connected by an idealized rotational joint AB. A marker fixated on segment B of this structure can now be observed to always be at the same distance to AB. In fact it is moving around on a sphere centered at the joint position, when A is used as a fixed reference. Assuming that AB is unknown this information can be used to estimate the joints position. An optimal joint between the two segments, thus, is placed on a position where the distance to the markers of the adjacent segments remain the same over the whole capture time. In other words, the variance of distance over time between the joint and the markers has to be minimized in order to optimize the joint position. The

variance of distance thus can be used as cost function. The minimization is done using a non linear optimization algorithm, which is explained in more detail later in this section.

This leaves open the question of how many markers are needed to perform the optimization. The problem can be visualized using intersecting spheres. The markers $m_1, m_2, ...$ on A and B are assumed to have the distances $r_1, r_2, ...$ to the joint AB. Considering one marker m_1 the joint position has to be somewhere on the sphere that has center $m_1$and radius r_1. Adding a second marker m_2 the joint is restricted to the circle created by the intersection of the markers spheres. Three markers in turn produce two points, where the joint can possibly be. Finally four markers enable us to determine the joint position exactly. Both segments, however, have to have at least one marker.

Using the variance as a cost function the topology of the skeleton model can be inferred. The value produced by the cost function for a (virtual) joint between two segments is called joint cost. To obtain the topology we now look at the segments and joints as being a graph, where the segments are the nodes and the joints are the edges. The joint costs is considered the edge weight. Therefore, the joint cost for all possible pairs of segments has to be calculated. The optimal skeleton then is the minimum spanning tree of the graph, which is inferred by the algorithm of Prim [Sed84]. This gives us a hierarchical tree as representation of the skeleton model.

Minimizing the joint cost for every pair of segments, however, is computationally very costly, which is why [KOF05] suggests an optimization for that matter. Instead of calculating the variance of distance over all frames only sample frames are used. Furthermore decreasing the precision by which joint positions are optimized can be reduced for that matter, reducing computation time even more.

3.5.1 The Global Minimization Function and its Derivative

The core of all these calculations is the cost function, a sum of variances, which has to be minimized. Therefore the function and its properties and parts are described here in more detail using a bottom-up approach. The variance is always evaluated between the markers of two clusters b_a and b_b, which in turn contain $|b_a|$ and $|b_b|$ markers. Only one joint c - between b_a and b_b - is considered at a time and its position in frame f (of N_f total frames) c_f. Furthermore the coordinates of c_f are xc_f, yc_f and zc_f, which are the variables of the cost function. Note that these coordinates have to be optimized for every frame in order to minimize the function. The position of marker n at frame f is given by $m_{n,f}$ and the number of frames for which a position of m_n exists $|m_n|$. $x_{n,f}, y_{n,f}$ and $z_{n,f}$ denote the coordinates of marker n at frame f.

Starting with the basic building block of the function, the average distance between a marker and the current joint is

$$\bar{d}(c, m_n) = \frac{1}{|m_n|} \sum_{f=1}^{|m_n|} \sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}$$

The variance in distance can then be written as

$$\sigma(c, m_n) = \frac{1}{|m_n|} \sum_{f=1}^{|m_n|} \left(\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + ...} - \bar{d}(c, m_n) \right)^2$$

Finally the joint cost is the sum of all the involved markers's variances divided by the number of markers:

$$jc(a, b) = \frac{1}{|b_a| + |b_b|} \sum_{m_n \in b_a \cup m_n \in b_b} \sigma(c, m_n) + \left[\alpha \cdot \bar{d}(c, m_n) \right]$$

The expression in brackets is suggested by [KOF05] to avoid that the algorithm finds the trivial solution, which would be placing the joint infinitely far away. A joint at infinity has a variance of zero, as does the optimal solution. Adding a penalty to the joint cost, that increases with the distance to the markers, thus, avoids this problem. α here serves as a weight determining the importance of the additional term. The penalty, however, is only important for the determination of the topology. When the position of a joint is to be found between segments, that are really connected, the algorithm is very unlikely to drift off to infinity. Good initialization values for the optimization also help avoiding this error as will be pointed out later.

Now that the joint cost has been defined the next step is to minimize it. Since the function depends nonlinearly on the parameters, the joint positions, sophisticated algorithms like Levenberg-Marquard or nonlinear gradient descent have to be used. According to [KOF05] these two iterative algorithms have about the same performance, in both, speed and accuracy. Since the nonlinear conjugate gradient is a little bit more straight-forward to implement, it is used in our system. Crucial for this algorithm, however, is a good initial guess for the joint positions. This becomes obvious considering the fact, that the joint positions are found recursively. So start values close to the optimum require far less iterations and avoid being trapped in local minima. For the estimation of the topology the center of gravity of two segment's markers is used as initialization for their in between joint. Once the topology is inferred a local optimization algorithm ensures good initialization for the precise calculation of the joints's positions.

The nonlinear conjugate gradient method, as the name suggests, uses gradient information for the optimization. Therefore the joint cost's first derivation has to be calculated for all parameters. First $\bar{d}(c, m_n)$ is derived, since it is needed for the partial derivation of the whole joint cost function. The derivation is demonstrated only for xc_f, but is almost the same for yc_f and zc_f.

$$\bar{d}'_{xc_f}(c, m_n) = \left(\frac{1}{|m_n|} \sum_{f=1}^{|m_n|} \sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2} \right)'$$

We are differentiating by xc_f for a certain f, which is only contained in a single summand of the sum give above Therefore the other summands are constants and their derivation 0. $\bar{d}'_{xc_f}(c, m_n)$ can thus be written as

$$= \left(\frac{1}{|m_n|}\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}\right)'$$

$$= \frac{1}{|m_n| * \sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}} * (xc_f - x_{n,f})$$

Then likewise the other coordinates can be calculated

$$\bar{d}'_{yc_f}(c, m_n) = \frac{1}{|m_n| * \sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}} * (yc_f - y_{n,f})$$

and

$$\bar{d}'_{zc_f}(c, m_n) = \frac{1}{|m_n| * \sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}} * (zc_f - z_{n,f})$$

The derivation of the variance then can be written as

$$\sigma'_{xc_f}(c, m_n) = \left(\frac{1}{|m_n|}\sum_{f=1}^{|m_n|}\left(\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2} - \bar{d}(c, m_n)\right)^2\right)'$$

$$= \left(\frac{1}{|m_n|}\left(\sqrt{(xc_1 - x_{n,1})^2 + (yc_1 - y_{n,1})^2 + (zc_1 - z_{n,1})^2} - \bar{d}(c, m_n)\right)^2\right)' + ...$$
$$+ \left(\frac{1}{|m_n|}\left(\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2} - \bar{d}(c, m_n)\right)^2\right)' + ...$$
$$+ \left(\frac{1}{|m_n|}\left(\sqrt{(xc_{m_n} - x_{n,m_n})^2 + (yc_{m_n} - y_{n,m_n})^2 + (zc_{m_n} - z_{n,m_n})^2} - \bar{d}(c, m_n)\right)^2\right)'$$

$$= -\frac{2}{|m_n|}\left(\sqrt{(xc_1 - x_{n,1})^2 + (yc_1 - y_{n,1})^2 + (zc_1 - z_{n,1})^2} - \bar{d}(c, m_n)\right) * \bar{d}'(c, m_n) -$$
...
$$+ \frac{2}{|m_n|}\left(\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2} - \bar{d}(c, m_n)\right) *$$
$$* \left(\frac{1}{\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}} * (xc_f - x_{n,f}) - \bar{d}'(c, m_n)\right) - ...$$
$$- \frac{2}{|m_n|}\left(\sqrt{(xc_{m_n} - x_{n,m_n})^2 + (yc_{m_n} - y_{n,m_n})^2 + (zc_{m_n} - z_{n,m_n})^2} - \bar{d}(c, m_n)\right) *$$
$\bar{d}'(c, m_n)$

because of the definition of $\bar{d}(c, m)$ we can reduce to

$$= \frac{2}{|m_n|}\left(\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2} - \bar{d}(c, m_n)\right) *$$
$$* \frac{1}{\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}} * (xc_f - x_{n,f})$$

$$= \frac{2}{|m_n|}\left(1 - \frac{\bar{d}(c,m)}{\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}}\right) * (xc_f - x_{n,f})$$

Then similarly for yc_f

$$\sigma'_{yc_f}(c, m_n) = \frac{2}{|m_n|}\left(1 - \frac{\bar{d}(c,m)}{\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}}\right) * (yc_f - y_{n,f})$$

and for zc_f

$$\sigma'_{zc_f}(c, m_n) = \frac{2}{|m_n|}\left(1 - \frac{\bar{d}(c,m)}{\sqrt{(xc_f - x_{n,f})^2 + (yc_f - y_{n,f})^2 + (zc_f - z_{n,f})^2}}\right) * (zc_f - z_{n,f})$$

Finally the derivation of the joint cost can be constructed by differentiating the parts

$$jc'_{xc_f}(a,b) = \frac{1}{|b_a|+|b_b|}\sum_{m_n \in b_a \cup m_n \in b_b} \sigma'_{xc_f}(c, m_n) + \left[\alpha \cdot \bar{d}'_{xc_f}(c, m_n)\right]$$
$$jc'_{yc_f}(a,b) = \frac{1}{|b_a|+|b_b|}\sum_{m_n \in b_a \cup m_n \in b_b} \sigma'_{yc_f}(c, m_n) + \left[\alpha \cdot \bar{d}'_{yc_f}(c, m_n)\right]$$
$$jc'_{zc_f}(a,b) = \frac{1}{|b_a|+|b_b|}\sum_{m_n \in b_a \cup m_n \in b_b} \sigma'_{zc_f}(c, m_n) + \left[\alpha \cdot \bar{d}'_{zc_f}(c, m_n)\right]$$

Using this formulas it is possible to differentiate by the joint coordinates xc_f, yc_f and zc_f for all frames f.

3.5.2 The Local Optimization Function and its Derivative

As is suggested earlier, the results of the topology estimation can be used to find good initialization values for the final joint calculation. This is an improvement to the algorithm described in [KOF05], where no specific initialization procedure is suggested. As an input my method uses the average distance $\bar{d}_{old}(c, m_n)$ between the joint c and the markers m_n, as it was found during the calculation of the topology. Then, for every frame(local), the initial position of the joint is set to a value where the square of difference between the actual and the average distance is minimized. Again nonlinear conjugate gradient method is used for the minimization and thus the cost function and its derivation are presented here.

Since the optimization is performed frame by frame and joint by joint only three parameters xc, yc and zc are needed for the joint coordinates. Before the local optimization, they are initialized to the center of gravity of the segments's $b'_a s$ and $b'_b s$ markers. The coordinates of the nth marker at the frame to be treated are given by x_n,y_n and z_n.

$$ljc(a,b) = \frac{1}{|b_a|+|b_b|}\sum_{m_n \in b_a \cup m_n \in b_b}\left(\sqrt{(xc - x_n)^2 + (yc - ...} - \bar{d}_{old}(c, m_n)\right)^2$$

$\bar{d}_{old}(c, m_n)$ is considered a constant for the differentiation, therefore the derivative of the local joint cost function is:

$$ljc'_{xc}(a,b) = \frac{2}{|b_a|+|b_b|} \sum_{m_n \in b_a \cup m_n \in b_b} \left(\left(1 - \frac{\bar{d}_{old}(c,m_n)}{\sqrt{(xc-x_n)^2+(yc-...}} \right) * (xc - x_n) \right)$$

Before implementing this approach I tried triangulation (lateration to be precise; see [Val06] for more details), to generate good initialization values. As was pointed out earlier at least four markers $m_1, m_2...$ and their distances to the joint $r_1, r_2...$ are needed to precisely specify the joints's position. Using trilateration the joint can easily be inferred. In theory and as well in most test case it works very well. For frames, however, where the distance between two markers m_x and m_y of different segments is bigger than the sum of r_x and r_y, problems arise. Since the spheres drawn by $r_1, r_2...$ are required to intersect at least at one point the algorithm finds no useful solution. This can be fixed by continually increasing the radii until an intersection can be calculated, which however drastically decreases accuracy and thus makes the whole optimization pointless. The big advantage of the lateration method over the least squares approach is that it has a closed form solution. Least squares, however, proves more stable, especially for measured data, and is therefore used.

3.6 Nonlinear Conjugate Gradient Method

The nonlinear conjugate gradient Method(NLCG) is an algorithm that can be used to iteratively solve optimization problems. To make the method applicable the problem has to be expressed as a function $f(x)$. This function as well as the gradient $f'(x)$ has to be continuous. In our case NLCG is used to find the minimum of the function and only this case will be considered here, although calculating the maximum is not very different.

The description of NLCG here is kept as short as possible and much of the mathematical background is therefore left away. This chapter is only intended to present the idea of NLCG. For mathematical proofs and more details on this topic please refer to [She94] and [PTVF92]. Before getting started, nevertheless, some terms have to be defined. Then the steepest descent and the conjugate direction/gradient methods are presented to show the initial concepts, from which NLCG is derived.

The Gradient

Given a function $f(x)$, where x is a n- dimensional point. The gradient of f is defined as the vector holding the partial derivatives, thus:

$$f'(x) = \nabla f(x) = \begin{bmatrix} \frac{\partial}{\partial x_1} f(x) \\ \frac{\partial}{\partial x_2} f(x) \\ \cdots \\ \frac{\partial}{\partial x_n} f(x) \end{bmatrix}$$

For a point x the gradient gives the direction of greatest increase.

The Quadratic Form

A quadratic form according to [She94] is defined as "a scalar, quadratic function of a vector with the form

$f(x) = \frac{1}{2} x^T A x - b^T x + c$, where A is a matrix, x and b are vectors and c is a scalar constant."

As shown in [She94] the gradient is then $f'(x) = Ax - b$

Linear Independence

In linear algebra a set of vectors $\{v_1, v_2, v_3, ...v_n\}$ is called linearly independent if no vector of the set can be constructed by a linear combination of the other vectors. In other words $a_1 * v_1 + a_2 * v_2 + ... + a_n * v_n = 0$ for all scalars $a_1, a_2, ...a_n$.For a more detailed description of linear independence please refer to [Mey00].

Line Search

Given a function $f(x)$, where x is a n- dimensional point and r is a n- dimensional vector. Starting from a point x_0 line search is set to find the α for which $f(x_0 + \alpha * r)$ takes on a minimum value. For NLCG some different algorithms have been suggested, e.g. Brent's algorithm by Press et al. in [PTVF92] or Newton-Raphson/Secant by Shewchuk in [She94]. In our system Brent's algorithm is implemented, because it spares us from having to deal with the second order derivative of the function described in section 3.5.1. For the rest there is not much difference between the two algorithms. Both iteratively try to find at least a local minimum and are limited in precision only by round-off errors, which is more then sufficient. Shewchuk in [She94] even suggests to use a fast and inexact line search in order to save calculation time that can better be invested at another point in the NLCG algorithm.

3.6.1 The Method of Steepest Descent

The method of steepest descent, which is sometimes also called gradient descent method uses gradient information to find the (local) minimum of a function. Starting at an arbitrary point $x_{(0)}$ this approach always follows the direction of the steepest descent to iteratively find the minimum.

Since the direction of the greatest increase is given by the gradient the so-called *residual* $r_{(i)} = -f'(x_{(i)})$ describes the direction of steepest descent. The algorithm therefore takes a step in the direction of $r_{(i)}$. For fast convergence it is now important to set the step size α to a value that evaluates to a minimum of f. The function can now be seen as a one-dimensional function $f(\alpha)$. Since a continuous function has its extremes at a value where the first derivative is 0, we are setting $\frac{\partial}{\partial\alpha}f(x_{(i)})$ equal to 0. Following [She94] it can be shown that $\frac{\partial}{\partial\alpha}f(x_{(i)}) = f'(x_{(i)})^{T}\frac{\partial}{\partial\alpha}x_{(i)} = f'(x_{(i)})^{T}r_{(i-1)}$. The two vectors $f'(x_{(i)})^{T}$ and $r_{(i-1)}$ have to be orthogonal to equal 0. This fact can be used as a criterion to find the optimal α. In case the function $f(x)$ is given in quadratic form and the matrix A is known there is a closed form solution for α. Otherwise α has to be found with *line search*. The α calculated is then used to find the next position by $x_{(i+1)} = x_{(i)} + \alpha r_{(i)}$. Then again the residual $r_{(i+1)}$ and the new α can be evaluated for $x_{(i+1)}$. This is then repeated until some convergence criterion is reached. Figure 3.2 shows the convergence of the steepest descent method finding the minimum of a paraboloid.

To sum things up the steepest descent algorithm looks like this:

- Choose some starting point $x_{(0)}$
- While the number of iterations is under some limit
  - Calculate the residual $r_{(i)}$ at current position $x_{(i)} : r_{(i)} = -f'(x_{(i)})$
  - Find α that minimizes $f(x_{(i)} + \alpha r_{(i)})$
  - Compute new position $x_{(i+1)} = x_{(i)} + \alpha r_{(i)}$

3.6.2 Conjugate Directions

From figure 3.2 we can see that steepest descent takes a lot of steps into the same directions. This of course can not be considered optimal. Instead we would want to take exactly one step into each of the orthogonal directions and then arrive at the minimum. Unfortunately finding the minimum this way like depicted in figure 3.3 is impossible without knowing the minimum in advance. This now is where conjugacy and the method of conjugate directions comes into play.

Considering two vectors $d_{(i)}$ and $d_{(j)}$ and a square matrix A. The criterion, which has to be satisfied in order to make the vectors *A-orthogonal* or *conjugate* is:

$$d_{(i)}^{T} A d_{(j)} = 0$$

For a n- dimensional quadratic form n conjugate search directions $d_{(i)}$ are needed to find the minimum in n steps. The mathematical proof for this and the algorithm for calculating such a set of conjugate directions - the so-called conjugate Gram–Schmidt process - can be found in [She94].

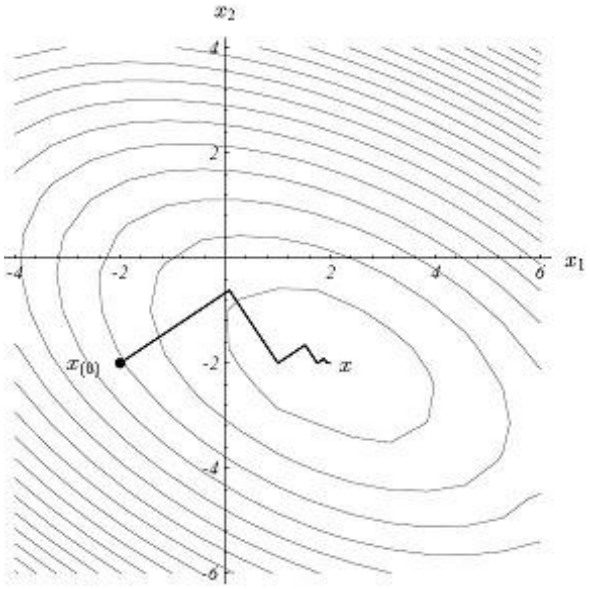

Figure 3.2: Topview of a paraboloid and the steps taken by the steepest descent method (zig-zag line) from [She94]

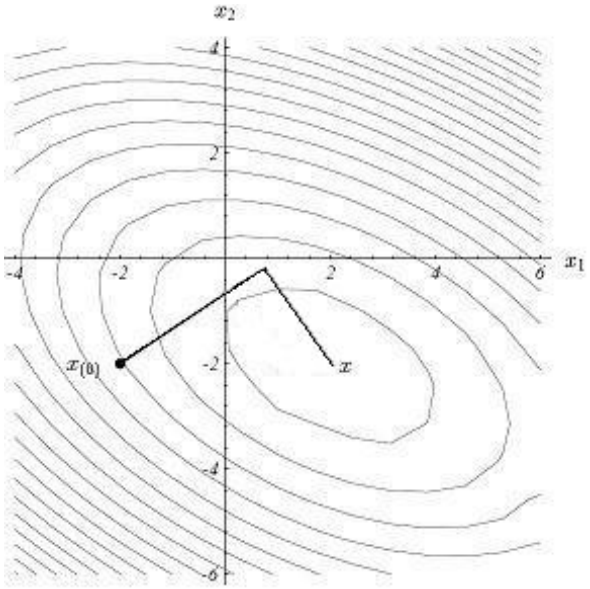

Figure 3.3: Topview of a paraboloid with an optimal(and unfortunatly impossible) two-step finding of the minimum

To calculate the search directions the Gram-Schmidt conjugation needs n linearly independent vectors as an input. This vectors are then made conjugate. The drawback of this approach however is that Gram-Schmidt conjugation has to keep all directions in memory and needs $O(n^3)$ operations, which might be a rather large number for a lot of optimization problems. In our case, for the optimization of the joint positions, n is a four-, or for larger motion-sequences even a five-figured, number. These problems are finally solved in the conjugate gradient algorithm, which is a special case of the conjugate direction method.

3.6.3 Conjugate Gradients

For the conjugate gradient method the residuals are used to calculate the search directions with the Gram-Schmidt conjugation. A residual is always orthogonal to the previous search directions as well as the previous residuals. Therefore residuals meet the requirement of being linearly independent. For the mathematical proof please refer to [She94]. The conjugate gradient method thus is a combination of steepest descent and conjugate direction inheriting the best properties from both algorithms. Therefore only n steps are needed to find the minimum. In addition, due to the special orthogonality-attributes described at the beginning of the section, the search directions of a step can be calculated using only the residual of the current and the previous step. This deprives us from having to store all the residuals and incredibly speeds up the process. According to [She94] for quadric forms "space complexity and time complexity per iteration are reduced to $O(m)$, where m is the number of nonzero entries of A".

3.6.4 The Method of Conjugate Gradients for General Continuous Functions

As already indicated before the conjugate gradient method can be used to find the minimum of any continuous function $f(x)$ as long as the gradient $f'(x)$ can be computed and is then usually referred to as the *nonlinear conjugate gradient method*. The only difference to having a quadratic form is that it might not converge as nicely or in some cases at all to the minimum point. After the description of the algorithm in the next section follows a more detailed analysis of the convergence. This is important to understand why it is not possible to calculate a perfect solution to our minimization problem and the joint positions can not be found in a single step. Therefore measures like the local optimization described in section 3.5.2 are taken to further improve the results. Also it is essential to find the skeleton's parameters and fit the skeleton back to the data to eliminate outliers in the joints' positions.

The Algorithm of the Nonlinear Conjugate Gradient Method

Starting at some point x_0 the first search direction d_0 is initialized to the residual $r_0 = -f'(x_0)$. Then the NLCG-algorithm looks like this:

1. Minimize $f(x)$ along the search direction of d_i starting at x_i. In other words find the α_i that minimizes $f(x_i + \alpha_i * d_i)$.

2. Calculate the new position $x_{i+1} = x_i + \alpha_i * d_i$.

3. Compute the new residual as the negation of the gradient at the new position: $r_{i+1} = -f'(x_{i+1})$.

4. Now the residual r_{i+1} has to be made conjugate to the old search directions using the Gram-Schmidt method. The new search direction is produced using the formula: $d_{i+1} = r_{i+1} + \beta_{i+1} * d_i$. For β_{i+1} multiple choices are suggested throughout the literature. The most prominent ones are Hestenes-Stiefel, Fletcher-Reeves and Polak-Ribiere. The later two are considered the more reliable and therefore used in our system. For an evaluation of the results produced by the two algorithms please refer to chapter 5. A discussion on the convergence of NLCG using the two methods can be found in the next section. Fletcher-Reeves(FR) and Polak-Ribiere(PR) use the following formulas to calculate β_{i+1}.

   - FR: $\beta_{i+1} = \frac{r_{i+1}^T * r_{i+1}}{r_i^T * r_i}$
   - PR: $\beta_{i+1} = \frac{r_{i+1}^T * (r_{i+1} - r_i)}{r_i^T * r_i}$

5. Go to 1 if the maximum number of iterations is not reached yet and the difference between x_{+1} and x_i is larger than a certain threshold.

Convergence of NLCG

The main problem with minimizing general functions is, that they might have multiple local minima. In that case the NLCG method may converge towards one of these instead of the global minimum. Even worse, if the function has no lower bound, the algorithm might not even find a local minimum. For that reason it is advisable to start with a position close to the global minimum in case that is possible. In our system relatively good initialization values can be produced as described in section 3.5.2. In general Fletcher-Reeves is said to converge if the start position is close to the minimum, while Polak-Ribiere converges faster, while it might in some rare worst cases get stuck.

Another problem is, that the search directions easily loose conjugacy due to round-off errors and when the minimized function is different from a quadratic function. Shewchuk therefore suggests in [She94] to restart the algorithm every n iterations (i.e. resetting the search direction to the current residual as it is done in the initialization step described in the previous section). In our case, however, this is not possible since processing of even n iterations would take an enormous computational effort.

For most cases the precision of the calculated minimum position nevertheless is more than sufficient. Additionally, for the calculation of the joint positions in our system, the results are further improved using inverse kinematics as described in chapter 2.6.

3.7 Closed-Form Solution of Absolute Orientation Using Unit Quaternions

The method presented in this section was developed by Horn and is described in more detail in [Hor87]. It is intended to transform one coordinate system to another given measurements of the same points in both systems. Initially the method was developed for the photogrammetric problem of matching two pictures taken from different perspectives. Therefore besides orientation and translation also a scale parameter can be determined. For our purposes, which are skeleton parameterization and a hierarchical inverse kinematics algorithm scale is not needed and is therefore left away.

As the heading suggests the algorithm uses unit quaternions to represent rotations. Therefore at least some understanding of quaternions and their properties is advised. A short introduction to quaternions is given in section 2.7.

Since the algorithm is intended for measuremet-data (like our marker-positions) we are not expecting to find a transformation that perfectly matches the two coordinate systems. The method instead minimizes the square of residual errors. Numerous other algorithms like NLCG (section 3.6) can be used to achieve this goal. The advantage of the method described here is that it finds a closed-form solution instead of iteratively approximating the minimum. Thus it is faster, more robust and one doesn't have to think about finding a start value. Also the number of the measurements incorporated in the calculations is arbirtrary as long as there is more then two.

3.7.1 Finding the Optimal Translation and Rotation

Given are two coordinate systems called left and rigth here as well as measured coordinates of n points. The coordinates are denoted $\{r_{l,i}\}$ and $\{r_{r,i}\}$. The transformation that is to be found can be written as $r_r = R(r_l) + r_0$, where r_0 is the translation and R is the rotation matrix. Since there can't be found a perfect transformation, unless for perfect data, there will almost always be a residual error $e_i = r_{r,i} - R(r_{l,i}) - r_0$. The sum over all points of the squares of these errors is now to be minimized. Thus we are looking for the minimum of $\sum_{i=1}^{n} \|e_i\|^2$.

3.7.2 The Translation

To find the translation first the averages of all measurements in both coordinte systems - the so-called centroids - have to be calculated. They are defined as:

$$\overline{r}_l = \frac{1}{n}\sum_{i=1}^{n} r_{l,i} \text{ and } \overline{r}_r = \frac{1}{n}\sum_{i=1}^{n} r_{r,i}$$

The measured points now can be expressed relative to the centroid by $r'_{l,i} = r_{l,i} - \overline{r}_l$, $r'_{r,i} = r_{r,i} - \overline{r}_r$and the translation by $r'_0 = r_0 - \overline{r}_r + R(r'_{l,i})$

The sum of squared errors, that we want to minimize, can now be written as

$$\sum_{i=1}^{n} \|e_i\|^2 . = \sum_{i=1}^{n} \left\| r'_{r,i} - R(r'_{l,i}) - r'_0 \right\|^2$$

This term can also be written as

$$\sum_{i=1}^{n} \left\| r'_{r,i} - R(r'_{l,i}) \right\|^2 - 2r'_0 \sum_{i=1}^{n} (r'_{r,i} - R(r'_{l,i})) + n \|r'_0\|^2$$

Since the measurements are relative to the centroids the middle term is 0. This leaves

$$\sum_{i=1}^{n} \left\| r'_{r,i} - R(r'_{l,i}) \right\|^2 + n \|r'_0\|^2$$

This means that the error is minimized where $r'_0 = 0$. Thus $r_0 = \overline{r}_r - R(\overline{r}_l)$. In other words the translation is optimally defined as the difference of the right centroid and the rotated left centroid. This difference can be dealt with, after the rotation has been determined.

3.7.3 The Rotation

To find the optimal rotation the error-term is expanded further. ($n \|r'_0\|^2$ is eliminated since it will be 0 in the end)

$$\sum_{i=1}^{n} \left\| r'_{r,i} \right\|^2 - 2\sum_{i=1}^{n} r'_{r,i} \cdot R(r'_{l,i}) + \sum_{i=1}^{n} \left\| R(r'_{l,i}) \right\|^2$$

Since the first and the third term of the expression don't change with the rotation only the second term is left to optimize. The sign of the second term is negativ, which leaves as the following term to maximize in order to find the optimal rotation R:

$$\sum_{i=1}^{n} r'_{r,i} \cdot R(r'_{l,i})$$

Now the problem is reformulated using quaternions. Knowing that the rotation can be performed by multiplication with a unit quaternion and that the dot-product is commutative the expression can be rewritten as

$$\sum_{i=1}^{n} (\overset{\circ}{q}\overset{\circ}{r}'_{l,i}\overset{\circ}{q}^{*}) \circ \overset{\circ}{r}'_{r,i}$$

. The goal now is to find the $\overset{\circ}{q}$ for which the term is maximized. Following $QP3$ from 2.7 the former expression is equivalent to

$\sum_{i=1}^{n} (\overset{\circ}{q}\overset{\circ}{r}'_{l,i}) \circ (\overset{\circ}{r}'_{r,i}\overset{\circ}{q})$, where expansion of $\overset{\circ}{r}'_{l,i}$ and $\overset{\circ}{r}'_{r,i}$ to $4x4$ matrices leads to

$\sum_{i=1}^{n} (\bar{R}^{T}_{l,i}\overset{\circ}{q}) \circ (R_{r,i}\overset{\circ}{q})$ and $\sum_{i=1}^{n} \overset{\circ}{q}^{T}\bar{R}^{T}_{l,i}R_{r,i}\overset{\circ}{q}$ which equals

$$\overset{\circ}{q}^{T}\left(\sum_{i=1}^{n} \bar{R}^{T}_{l,i}R_{r,i}\right)\overset{\circ}{q}$$

By defining a $4x4$ matrix $N = \sum_{i=1}^{n} N_i$ where $N_i = \bar{R}^{T}_{l,i}R_{r,i}$the problem can be finally written as:

$$\overset{\circ}{q}^{T} N \overset{\circ}{q}$$

3.7.4 Calculating matrix N

The matrix N is calculated from sums and products of the n measurement's coordinates.($\overset{\circ}{r}'_{l,i} = ((x'_{l,i}, y'_{l,i}, z'_{l,i}), 0)$ and $\overset{\circ}{r}'_{r,i} = ((x'_{r,i}, y'_{r,i}, z'_{r,i}), 0)$) For reasons of efficiency and clarity a helper-matrix M can be used. It is a $3x3$ matrix and defined as

$$M = \begin{bmatrix} S_{xx} & S_{xy} & S_{xz} \\ S_{yx} & S_{yy} & S_{yz} \\ S_{zx} & S_{zy} & S_{zz} \end{bmatrix}$$

where

$$S_{xx} = \sum_{i=1}^{n} x'_{l,i} x'_{r,i} \text{ and } S_{xy} = \sum_{i=1}^{n} x'_{l,i} y'_{r,i} \text{ and so forth.}$$

The elements of matrix N can then be computed as sums of elements of matrix M.

$$N = \begin{bmatrix} (S_{xx} + S_{yy} + S_{zz}) & S_{yz} - S_{zy} & S_{zx} - S_{xz} & S_{xy} - S_{yx} \\ S_{yz} - S_{zy} & (S_{xx} - S_{yy} - S_{zz}) & S_{xy} + S_{yx} & S_{zx} + S_{xz} \\ S_{zx} - S_{xz} & S_{xy} + S_{yx} & (-S_{xx} + S_{yy} - S_{zz}) & S_{yz} + S_{zy} \\ S_{xy} - S_{yx} & S_{zx} + S_{xz} & S_{yz} + S_{zy} & (-S_{xx} - S_{yy} + S_{zz}) \end{bmatrix}$$

3.7.5 Finding the optimal $\overset{\circ}{q}$

Now that the problem has been defined as $\overset{\circ}{q}^T N \overset{\circ}{q}$ the next step is to find the unit quaternion $\overset{\circ}{q}$, which maximizes it. Horn proofs in [Hor87] that $\overset{\circ}{q}^T N \overset{\circ}{q}$ is maximal, when $\overset{\circ}{q}$ is set to the eigenvector of N, which has the greatest eigenvalue. Therefore, first the greatest eigenvalue has to be found and then the corresponding eigenvector can be computed. The eigenvalues are the roots of the characteristic polynomial of N. The coefficients of the characteristic polynomial can be calculated by $\det(N - \lambda I)$ as described in 2.8.

The characteristic polynomial of N is of fourth order - a so-called quartic - since N is a $4x4$ matrix. The roots of a quartic can be found closed-form using Ferrari's method. Substituting the largest positive λ the linear equation system $(N - \lambda I)x = 0$ is solved to get the eigenvector corresponding to λ. This is done using Gaussian elimination, which brings $(N - \lambda I)$ into echelon form and then calculates the components of the vector starting at the lowest row and then substitutes back into the upper rows.

Finally the optimal $\overset{\circ}{q}$ can be derived from the eigenvector. It is the unit quaternion, that points in the same direction as the eigenvector. From the quaternion then the rotation matrix can be calculated using the formulas presented in 2.7.

3.7.6 The Algorithm

Putting the things from the last sub-sections together the algorithm can be summed up in the following steps.

1. Calculate the centroids of the two coordinate system $\overline{r}_l$ and $\overline{r}_r$as the average of the measured points.

2. Express the measured coordinates relative to the centroids $r'_{l,i} = r_{l,i} - \overline{r}_l$, $r'_{r,i} = r_{r,i} - \overline{r}_r$

3. From $r'_{l,i}$ and, $r'_{r,i}$ calculate the matrix M
4. Use the elements of M to produce N.
5. Compute the determinant $\det(N-\lambda I)$ in order to retrieve the characteristic polynomial of N
6. Find the roots of the characteristic polynomial and select the largest positive eigenvalue λ.
7. Replace λ in $(N - \lambda I)x = 0$ and solve for vector x.
8. Calculate unit quaternion $\overset{\circ}{q}$ that points in the same direction as x.
9. Derive rotation matrix R from $\overset{\circ}{q}$.
10. Finally compute the translation vector $r_0 = \overline{r}_r - R(\overline{r}_l)$.
11. A measured coordinate $r_{l,i}$ from the left coordinate system can now be transformed to the right coordinate system using the expression $R(\overline{r}_{l,i} - \overline{r}_l) + r_0$

Note that the rotation R can also be performed using quaternion products or another representation. In our system, however, for transformations $4x4$ matrices are used and thus have been emphasized here.

3.7.7 Minimum Spanning Tree

After calculating an approximate cost of putting a joint between two limbs, the next step is to decide which joints are to be kept for the final skeleton structure. This problem can be easily modeled using an undirected weighted graph(like the one displayed in figure 3.4. Here the limbs are the nodes and joints are the edges. The joint cost calculated in the previous step is used as edge weight. Since the edge weight has to be stored for all possible pairs of nodes an adjacency matrix is the best choice to store them.

Our goal now is to sort out all edges/joints, which have a joint cost too high to coincide with a joint on the real skeleton. At the same time it is important that the skeleton remains connected. This goal can be achieved by finding the minimum spanning tree (MST) of the graph described before. An MST of a weighted graph is defined as the set of edges that connects all nodes, so that the sum of the edge-weights is at least as small as the sum of edge-weights of any other set of edges that connects all nodes. For the calculation of MSTs many algorithms have been developed. Most of them are based on the property that with any partitioning of a graph into two sets the MST always contains the shortest of the edges, which connect one set with the other. This can be easily proven as is demonstrated in [Sed84].

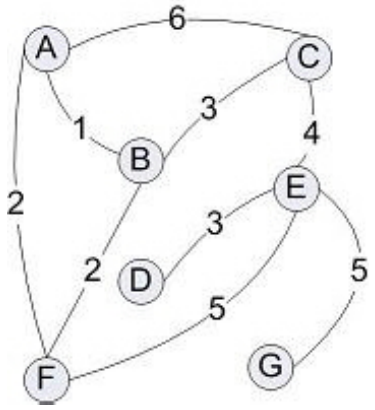

Figure 3.4: Example of an undirected weighted graph

Using the above property an MST can be generated by starting at an arbitrary node and connecting it to the node, which it is closest to. In other words the edge with the smallest weight, which connects nodes already in the tree to nodes outside the tree is added to the MST. In case two or more edges are having the same weight one can be chosen arbitrarily, thus different MSTs can be generated. In our case, however, the MST should be unique and ideally coincide with the real skeleton.

For our system the algorithm of Prim is used to find the MST, because it is the method of choice for dense graphs. The computation time of the MST, however, is very short compared to the rest of the calculations. Since the number of nodes won't be much more than 15 for a human skeleton the algorithm used is not decisive for the overall performance of the system. In an additional step, after the algorithm of Prim has been executed, the root of the MST is set to the node with the most connected edges. This is done to improve the result of the inverse kinematics routine later on. The advantages of this approach are described in section 2.6.

The algorithm of Prim

The algorithm of Prim is illustrated in figure 3.5. It works on three sets of nodes: tree-nodes(circles), fringe-nodes(squares) and invisible nodes(grey circles). Each node in the fringe has a so-called priority, which is the smallest edge cost of the edges connecting the fringe-node to the tree. At the start of the algorithm one node is added to the tree, which can be arbitrarily selected. The rest of the nodes are marked invisible, while the fringe is empty. The following steps are executed "number of nodes -1" times:

1. Any invisible nodes which are connected to the node last added to the tree are added to the fringe.

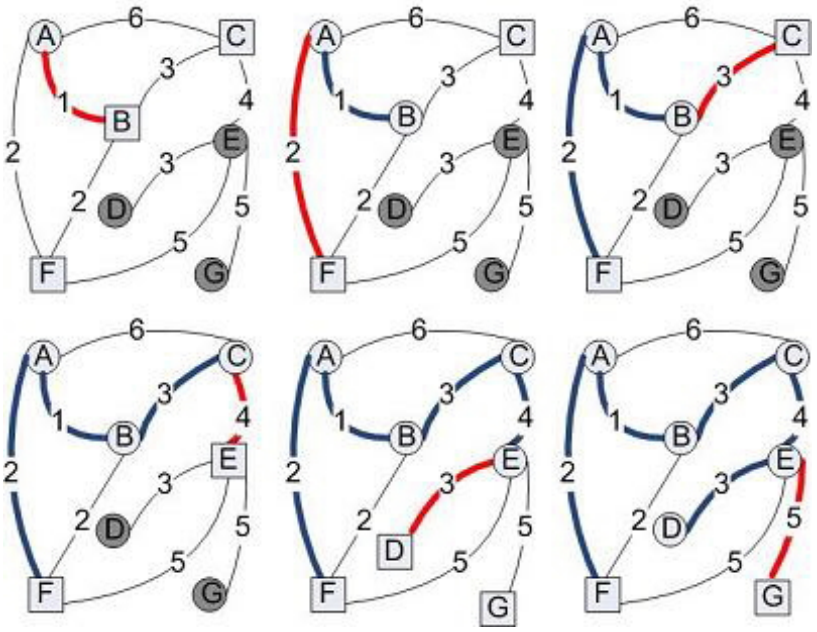

Figure 3.5: Steps taken by the algorithm of Prim in a sample graph displaying the different node types throughout the process tree-nodes(circles), fringe-nodes(squares) and invisible nodes(grey circles). Edges added to the MST are shown red/blue.

2. For each node in the fringe check, whether it has edges connecting it to the last node. If there are edges and if the edge-weight of one of these edges is smaller than the nodes priority, make the smallest edge-weight priority.

3. Find the edge that connects the tree-set with the fringe-set that has minimum weight. In other words we are looking for the edge between the tree and the node in the fringe with the lowest priority value.

4. Add the edge and the fringe-node at the end of the edge to the tree(and out of the fringe-set).

Finally the tree-set contains all nodes and a set of edges, where a path between any pair of nodes exists. The other sets should be empty at this point.

3.8 Fitting a Rigid Body Skeleton

After the joint positions have been optimized two joints connected to a single limb can still vary in distance. Thus the length of a limb is not fixed. This phenomenon can have a number of reasons:

- Noise in the input data due to the inaccuracy of the tracking system

- NLCG algorithm gets trapped in local minimum for some frames or other errors introduced by the iterative nature of this algorithm.
- Human joints are only approximate rotational joints.
- Markers move relative to the limb due to skin motion etc.

Then again, it might be important to have a skeleton with rigid segments, because

- It provides a base, which can be used to compute an estimate for the noise contained in the data.
- For performance animation usually a rigid body skeleton is needed.
- For performance animation smooth movements are more important then inaccuracies, which might be introduced for some frames.
- Parameterization allows measurement of error.
- It for many cases improves accuracy.

For this reasons a skeleton with rigid segments is created and then fitted back to the data.

3.8.1 Finding the Parameters

The idea is to collect as much useful marker and joint data as possible over all frames. Then the sections of the skeleton are lined up individually and marker and joint data is averaged over multiple frames. These averages are then used in a second step to put the skeleton back together. The length parameters that can be inferred from the newly assembled skeleton are the best compromise between the measurements of all frames.

The Algorithm

In order for the algorithm to work at least one frame is required to have not missing markers at all. The first frame for which this is true is considered the reference frame f_r. The data is treated limb by limb for the first part of the algorithm. Thus only the marker positions of one segment and the positions of the attached joints are considered at a time. The set of marker/joint positions of the current limb in the ith frame f_i is denoted l_i. For all limbs the following steps have to be processed

1. Iterate over all frames and find the set of limb-frame data $L = \{l_1, l_2 ... l_n\}$, where no markers of the current limb are occluded. Positions of markers and joints will be treated equally in the following and denoted with x'_{ji}

for the jth marker/joint in l_i. Thus $l_i = \{x'_{1i}, x'_{2i}, ...x'_{mi}\}$, where m is the number of markers and joints of the limb. These positions are no absolute values but seen relative to the centroid as defined in section 3.7. Here the centroid of l_i is $\overline{c}_i = \frac{1}{m}\sum_{j=1}^{m} x_{ji}$,where x_{ji} are the absolute positions. (Thus $x'_{ji} = x_{ji} - \overline{c}_i$)

2. The method described in section 3.7 is used to find the rotation and translation $T_i(l_i)$ that best aligns the limb l_i (i.e. its marker and joint positions x'_{ji}) with the position it has in the reference frame l_r. (thus $T_i(l_i) \approx l_r$) Note that connections between limbs are broken up and therefore after the transformation the skeleton for one joint has two different positions for both adjacent limbs.

3. For every marker or joint now a cloud of positions C_j exists with $C_j = \{T_i(x'_{j1}), T_i(x'_{j2}), ..T_i(x'_{jn}))$

4. The coordinates of C_j are assumed to be approximately normally distributed around the average of C_j. Therefore we can use statistical means to identify outliers. For that reason the average position $\bar{X}_j$and the standard deviation $\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(\bar{X}_j - T_i(x'_{ji}))^2}$are calculated. All x'_{ij} which are farther from $\bar{X}_j$ than the standard deviation σ are not considered for the following calculations. Thus a new average position $\bar{X}'_j$is calculated from the data within the standard deviation. This average position is used as an offset in the second part of the algorithm. Additionally the average position of the centroids $\overline{c}_i$ is calculated and denoted $\overline{c}$.

In the second step the offsets $\bar{X}'_j$ and the information about the topology are used to compose a skeleton in reference position. As mentioned in section 3.7.7 the root of the hierarchical skeleton structure most probably is the torso. In the following a joint on a limb will be called inner joint, if in the hierarchy it is closer to the root, while the other joint(s) will consequently be called outer joint. Note that the root only has outer joints and every other limb has exactly one inner joint and arbitrary outer joints.

For the outer joints and markers of the root the offsets given by the corresponding $\bar{X}'_j$ can be used directly to set the positions. So the absolute position of the joint/marker belonging to $\bar{X}'_j$ can be calculated as the sum of the root's centroid and the offset: $\overline{c}_{(root)} + \bar{X}'_j$.

For limbs other than the root things are a little more complicated. Starting with the limbs adjacent to the root we have to work our way through the hierarchy. The following steps have to be taken to add a limb to the skeleton:

1. Select the joint corresponding to the inner joint of the limb. This joint has to be an outer joint of a limb already added, thus the absolute position has already been calculated.
2. Since we know the offset between the inner joint and the centroid we can calculate the absolute position of the limb's centroid ($\overline{c}_{(current)}$).
3. Using the offsets $\bar{X}_j'$ we can now calculate the absolute positions of the limb's markers and outer joints by $\overline{c}_{(current)} + \bar{X}_j'$.

After these steps have been done for all limbs a reference skeleton has been created. This can be used to measure parameters like limb lengths and can be used for example for animation. Additionally it will be used to improve the accuracy of joint positions by fitting it back to the marker data in the next section.

3.8.2 Fitting the Skeleton Back to the Data

After having calculated a rigid section skeleton, its pose has to be adapted for each frame, so it fits the originally captured markers as good as possible. Therefore a process called inverse kinematics is used. An introduction to inverse kinematics can be found in section 2.6.

The skeleton calculated by the algorithms in the last sections has a high number of degrees of freedom. This is the drawback of having a flexible system, which adapts to different skeleton structures, because it doesn't allow us to put constraints on certain joints. If we had a skeleton model in advance, we could for example limit the knee to only one DOF. Without constraints fitting the skeleton as a whole is a lot more difficult. Since numerical algorithms would have to be used with all their drawbacks, I decided to use a different approach.

The skeleton is fitted to the data one limb at a time, which allows a closed-form solution. This in turn has the advantage of being very fast and robust, while the result still proves satisfactory as can be seen in chapter 5. Additionally intermediate results from the previous section can be used.

The idea is rather similar to that in the previous section. There the poses of the skeleton were changed into that of the reference frame in all frames. Now the joints of the skeleton in the reference frame's posture are rotated in a way, that the markers best match the originally tracked markers.

The Algorithm

First the root of our parameterized skeleton has to be matched with the root at frame i. In other words the transformation T_i has to be found, where $l_{root,i} \approx T_i(l_{root,r})$. Again the method of section 3.7 is used. For the other limbs the following calculations have to be made starting at the sections adjacent to the root.

1. First of all the offsets $\bar{X}_j^{'}$ are relative to the centroids. In this case, however, we need the offsets to be relative to the inner joint, since this is supposed to be the center of rotation. Therefore the offset to the inner joint is added resulting in $\bar{X}_j^{''} = \bar{X}_j^{'} + \bar{X}_{innerjoint}^{'}$. The set of these coordinates is now denoted $l_r' = \{\bar{X}_1^{''}, \bar{X}_2^{''}, ...\bar{X}_m^{''}\}$

2. For all frames the centroid $\bar{c}_i = \frac{1}{m}\sum_{j=1}^{m} x_{ji}$ has to be calculated. Then the offsets from the centroid $x'_{ji} = x_{ji} - \bar{c}_i$can be calculated. Finally $x''_{ji} = x'_{ji} + x'_{innerjoint,i}$ and l_i the set of coordinates of the ith frame relative to the inner joint is $l_i' = \{x''_{1i}, x''_{2i}, ...x''_{mi}\}$.

3. Now the rotation R_i is calculated, where $l_i' \approx R_i(\ l_r'\ )$. No translation is used, because we want the sections of the skeleton to stay connected.

4. After having calculated the rotation for the inner joint over all frames the absolute position of the outer joints can be calculated. Lets for the sake of simplicity of this discussion assume that the current limb has only one outer joint ($x_{outerjoint,i}$ ... outer joint in frame i). Then $x_{outerjoint,i} = x_{innerjoint,i} + x''_{outerjoint,i}$. Note that the position of the outer joint just calculated influences the centroid of the limb, which has this joint as inner joint. Thus it is evident, that the skeleton has to be fitted from the root down the hierarchy.

This way the joint positions and rotations can be calculated for all frames using a rigid body skeleton. This data can be easily used for animation employing forward kinematics as described in section 2.6.

Care has to be taken in case of occluded markers. In case a marker is missing in a frame it has no effect on the centroid $\bar{c}_i$. Therefore the centroid of the reference $\bar{c}$ also has to be calculated without that marker in order to allow a good match.

Chapter 4

Implementation

This chapter's main focus is the software implementation of the motion capture system. For the hardware implementation and the tracker software please refer to [PK07] and [Meh06]. The program is written using C++ using various libraries like `Qt`, `stl` and `newmat`. References to detailed descriptions of these can be found in the bibliography. Although Visual Studio was used I tried to keep the program platform independent by renouncing system specific libraries.

The program has four main subsystems, which to some extent can each be run independently given that the subsequent steps have already been executed before. These four parts are the temporal correspondence (section 4.2), the spectral clustering (section 4.3), joint estimation (sections 4.4, 4.5 and 4.6) and skeleton parameterization (section 4.7). The remainder of sections in this chapter is mainly about helper classes like those for I/O and the GUI. The order of the sections corresponds with the sequence in which the parts are executed during a normal program run starting off with the GUI.

4.1 The GUI

The graphical user interface is designed using `Qt` from Trolltech ([Tro07]) Therefore some basic knowledge of this library might be helpful for understanding this section. Since most of the implementation is already done in the `Qt` base classes the classes described here are rather small and contain only minor modifications.

Every GUI application that uses `Qt` needs one `QApplication` object, which in our case has been subclassed by **MyQApp** to allow modification of the exception handling. This is done in the overridden *notify* method, which catches events and outputs a message according to the exception that has been thrown. The **MyQApp** itself contains no visible elements, which is why it gets passed a **MainWindow** object.

4.1.1 The MainWindow Class

The **MainWindow**, which contains all the GUI elements is a subclass of `QMainWindow`. It contains two `QLineEdits` a `QPushButton` and five **OneStepWidgets**. We will get to the latter soon. The `QLineEdits` are used for inserting the number of markers and limbs. The `QPushButton` executes the whole procedure using the `slot` *executeButtonPressed* in **MainWindow.** The other methods are listed in the following.

- *setupGUI* places the GUI elements in the window and is called in the constructor.
- *clearParameters* and *loadParameters* reset all the parameters or load them from *config.xml.*
- *processTempCorr* starts the calculations that find the temporal correspondence of the markers.
- *processSpecClu* initiates the clustering of markers into limbs.
- *processJointFind* starts the processing of the joints and the skeleton parameters.

4.1.2 The OneStepWidget Class

OneStepWidget is a `QWidget` which is used to specify whether a processing step is to be taken (i.e. the *process...* methods of **MainWindow** executed) and where the input is to be taken from or the output written to. For this reason **OneStepWidget** has a `QLineEdit` to insert the path and file name. Also a `QPushButton` can be used to open a file dialog. Finally the `QCheckBox` can be used to prevent the processing of a step and instead refer to saved intermediate results.

4.2 Step 1: Temporal Marker Correspondence

This section deals with the implementation of the first preconditioning step, which has the task of giving the representation of a physical marker in the program an identifier. As described in section 3.3 finding this temporal correspondence is rather easy as long as all markers are visible during the entire capturing process. Markers, however, usually get occluded once in a while. Therefore I implemented some classes that execute the algorithms explained in section 3.3. These classes are **CorrespondenceFinder, PairClustering, MarkerSet** and **SpectralClustering**. The latter is documented in more detail in section 4.3. While the central processing of this stage is done by **CorrespondenceFinder**, it uses the others to perform certain operations or represent sub structures like **MarkerSet**.

`utils::mfml_container` *markersOrg*	Data structure that is used to load from the *mfml file.
`iod::data_container` *markers*	Container which stores the markers frame by frame. See section 4.10 for more details on this.
`std::map`<**unsigned short**,**MarkerSet**> *markerSets*	Maps the (preliminary) id of a virtual marker to the corresponding **MarkerSet.**
Matrix **distanceMatrix*	Contains the distances between the **MarkerSets.**

Table 4.1: Member variables of **CorrespondenceFinder**

CorrespondenceFinder has three ways of obtaining raw marker data. These modes are mapped onto three constructors, which either load the markers from file (*.mfml, *.csm) or generate a simple chain of bones, with some attached markers. The standard mode of operation is that of using *.mfml files, while the others can be used for testing purposes. Testing here is meant primarily for the following stages, because the algorithm that finds the temporal correspondence is not executed for these modes of operation. The ids of the markers are simply taken from file/generated and only the preprocessing for the spectral clustering is done. This preprocessing results in a matrix, which contains values that can be interpreted as cost of putting two markers on the same limb. Spectral Clustering is then used in the next processing stage to find the markers attached to a limb. Details on the file formats can be found in section 4.11 and 4.12.

The *member variables* of **CorrespondenceFinder** are listed in table 4.1.

As already mentioned **CorrespondenceFinder** has three constructors of which two are for testing purposes. The latter are rather similar and I will therefore describe them together in this paragraph. They both get passed an output file name. The constructor for the *.csm files additionally requires an input file name and the number of markers. At first the parameters are loaded from the *config.xml* file using the *setParameters* method. Then the marker positions are loaded from file/generated by the methods *load_csm_frames* or *generateData.* Since the ids are already determined the distance matrix can be calculated using the method *calculateCostMatrix4x4.* The matrix holds the variations of distance between the markers, which will be used in the next processing step to group the markers by limb. Finally the marker positions together with the marker ids are stored in the output file using *saveAll().*

For captured data the procedure is more complicated and the constructor has to be passed a certain parameter, which is later used for the clustering of the **MarkerSets**. After the parameters have been set and the marker positions have been loaded using *read_mfml* the algorithm described in the first part of section 3.3 is processed. This is done in *unifyIdentitiesMarkers* and finds the ids of markers over the periods where no marker gets occluded. (by using the

PairClustering class, which will be described later in this section) The problem with missing markers is solved in the method *reduceData*, which creates the **MarkerSets**. Subsequently the method *getDistMatrix* calculates the distances between the **MarkerSets.** Now the second algorithm described in 3.3 is executed and the **MarkerSets** are cojoined using the **SpectralClustering** class and the *combineClusteredMarkers* method. After that the markers are sorted by id for every frame using the method *sortMarkers.* This makes further processing faster and visual inspection of the output file easier. Finally again the cost matrix is calculated and together with the results written to file.

The methods of the **CorrespondenceFinder** are listed in the following by the order they are used during the processing of an *.mfml file. The rest of the methods, which are used primarily for testing purposes, will be documented in the following paragraphs.

- *setParameters* loads the parameters from *config.xml.*
- *read_ mfml* loads the content of the *.mfml file to *markersOrg.* The file contains the marker positions plus some attributes generated by the tracker. These are reprojection error, number of blobs from which they are reconstructed and a unique id for each marker in different frames. Most of these attributes are of no interest for the motion capture and therefore will be discarded by copying to *markers.*
- *unifyIdentitiesMarkers* gives the id of a marker in an old frame to the closest marker in the new frame. The method relies on the class **PairClustering** for this task. One instance of **PairClustering** is created for every frame except the first.
- *reduceData* in a first step iterates over all frames and markers and inserts all known marker positions into *markerSets.* Then all **MarkerSets** that are smaller than *MINRANGE* are erased and the updated **MarkerSets** rewritten to *markerSets.*
- *getDistMatrix* computes the distance matrix, which is used to find **MarkerSets** that belong to the same physical marker. The distance is calculated using the method *getMinDistTo* of the **MarkerSet** class, which implements the formula explained in section 3.3.
- *combineClusteredMarkers* rewrites the ids of markers, after the **MarkerSets** have been clustered. After the method has been executed the number of **MarkerSets** is reduced to the number of physical markers.
- *sortMarkers* arranges the markers of every frame in a way that the marker ids are in ascending order.

- *calculateCostMatrix* calculates the cost matrix on which the clustering is performed later on. As a cost function of two markers the standard deviation of the distance of those markers is used. Therefore a matrix containing the average distance is calculated first. In case *SAMPLING* is defined as zero all frames contribute to this average distance. Otherwise *SAMPLING* describes a step size and only samples are used for the calculation of the cost. Thus is recommended for longer capture sessions. Finally the matrix is written to an ASCII file using the method *saveStdDevMatrix.*
- *saveAll* writes the markers including the newly found id to a specified output file. Additionally for every frame the timestamp and number of markers are saved.

Finally the methods for test purposes:

- *createData* produces simple test data by simulating a three bone chain connected by two ball joints, which moves around a bit.
- *load_ csm_ frames* loads a csm file, which contains only points. These are interpreted as markers and stored in a **iod::data_container.**
- *simpleClustering* is a simple and fast clustering method that can be used instead of spectral clustering. The main disadvantage though is that it doesn't work if the clusters are too close to each other.

The purpose of the **PairClustering** class is to find a marker match for two successive frames. This means that for each marker in the first frame a marker in the second frame has to be found, which belongs to the same physical marker. The selection criteria for this process is the spatial proximity (for a detailed explanation of the algorithm please see section 3.3). Then the marker ids are transferred from the first to the second frame. In case no markers get occluded during the tracking process temporal marker correspondence can be established this way.

For the calculations **PairClustering** needs a couple of *member variables*, which are listed in table 4.2.

The constructor of **PairClustering** takes two frames (or to be more precise two iterators pointing to a `std::pair` of a **f_fsml** and a `std::vector` of **f_markers**) as arguments. Depending on the number of markers a custom sized matrix is created and filled with the distances between all possible matches of markers. Thereafter the actual match is determined using either *runClustering-MarkerNew* or *runClusteringMarkerLost.* Which method is used depends on the number of Markers in the frames. In case there is more or equally many markers in the second frame *runClusteringMarkerNew* is called and *runClusteringMarkerLost* else. Both methods set the array *clusterAssignment*, which is then used in *assignIds()* to transfer the marker ids to the second frame.

The methods used by the constructor are described in the following.

Matrix **A*	Matrix holding the distances between all markers of the first and the second frame.
u_int32_t *numberOfOldMarkers,numberOfNewMarkers*	Number of Markers in the first and the second frame.
u_int32_t **clusterAssignment*	Array that contains the matching markers. By indexing into the array using the marker id in the first frame the matching marker in the second frame can be found.

Table 4.2: Member variables of **PairClustering**

- *runClusteringMarkerNew* iterates over all entries in the distance matrix finds for each marker in the first frame a marker in the second frame that has minimal distance (in other words, we are looking for the minimum in each row of the distance matrix under the constraint, that no column is used twice. The result is the filled *clusterAssignment* array.
- *runClusteringMarkerLost* does the same as *runClusteringMarkerNew* except it searches matching markers for the points in the second frame.
- *assignIds* sets the ids of the markers in the second frame depending on the entries in *clusterAssignment.*

The subject of **MarkerSets** has already been touched on earlier in this section and section 3.3. As introduced there, a **MarkerSet** is a data structure holding a marker position over a set of successive frames. These positions are supposed to belong to the very same physical marker. Additionally some information about the relationship with other markers is contained as can be seen in the *member variables* in table 4.3.

MarkerSet is instantiated in **CorrespondenceFinder::***reduceData()* by passing a pointer to *markers* and the index of the marker to the constructor. There the *SAMPLINGMS* parameter is loaded and some initializations take place. The other methods of **MarkerSet** are itemized in the following list.

- *setRangeStart* and *setRangeEnd* sets the start and end indices of the frames, which are spanned by the **MarkerSet**
- *getRange* returns the number of frames spanned by the **MarkerSet.**
- *setMarkerIndex*, *getMarkerIndex, getStep, getNumOfSamples* return/set the values of the corresponding variables.
- *sample* initializes the *sampleFrames* array.

u_int64_t *rangeStart*	Index of the frame the **MarkerSet** starts with.
u_int64_t *rangeEnd*	Index of the frame the **MarkerSet** ends with.
u_int64_t *samples*	Parameter loaded from *config.xml*. It is used to determine how many frames are used for comparison with other **MarkerSets.**
u_int64_t **sampleFrames*	Array of indices that can be used to access the sample frames.
iod::data_container **markers*	Pointer to the **CorrespondenceFinder::** *markers* data structure. Used to index into for the sample frames so they don't have to be copied.
u_int64_t *step*	Determines how many frames are between two sample frames.
unsigned short *markerIndex*	Index of the marker the **MarkerSet** belongs to.
bool *sampled*	Shows if the **MarkerSet** has already been sampled.

Table 4.3: Member variables of **MarkerSet**

- *sampleAround* gets passed an index and fills the array *sampleFrames* with the indices around the parameter.

- *getSampleFrame* returns the frame with the index, which is passed as parameter.

- *getMinDistTo* computes the distance between the current **MarkerSet** and another one, passed as argument. This is done by finding the minimal distance between two frames of the two **MarkerSets** as expressed by the formula D_{ij} in section 3.3**.** To decrease processing time the frames are sampled and in a first step the minimal distance of the sample is calculated using the method *calculateDist.* Then, instead of sampling over the whole captured time span, only frames around the minimal sample are taken using *sampleAround* and *calculateDist* executed again. According to [KOF05] this delivers nearly the same results as comparing all frames in the first place.

- *calculateDist* calculates the distance between the sampled frames of the current **MarkerSet** and the one passed as argument. Therefore the method *getSampleFrame* is used to iterate over all the sample frames. The frame indices where the two sets match best are returned as reference parameters.

4.3 Step 2: Spectral Clustering

The class **SpectralClustering** implements the spectral clustering algorithm as it is described in 3.4. Thus it clusters a set of n objects/points $S = \{s_1, s_2, ..., s_n\}$ in $\mathbb{R}^l$ into k clusters.

SpectralClustering uses two special libraries. The first is `Newmat10`, which is a matrix library offering different matrix types and basic as well as more advanced matrix operations like eigenvalue decomposition. `Newmat10` is optimized to work with large matrices (i.e.: matrices with over hundred entries). This is very useful since the input to the spectral clustering algorithm is a distance matrix containing the distance values between the markers, which results in about thousand entries. SpectralClustering uses the **Matrix**, **SymmetricMatrix** and **DiagonalMatrix** classes, its basic operations and the *Jacobi* decomposition. Please refer to [Dav06] for a detailed documentation of these classes and the `Newmat10` library. Furthermore Meyer describes in [Mey00] how the Jacobi method for eigenvalue decomposition works.

The second library used is the `Kmlocal` package in the version 1.7.1. This library contains classes that can be used to perform k-means clustering as needed in step 5 of the spectral clustering algorithm presented in 3.4.2. `Kmlocal` implements Lloyd's algorithm for k-means clustering as well as a local search heuristic. Also a hybrid approach combining these two methods is available. Documentation of kmlocal and a discussion on k-means clustering can be found in [KMN+02b], [KMN+02a] and [DGK04].

The constructor of **SpectralClustering** takes the dimension and number of the points as well as the number of clusters as arguments. The other important methods are described in the next paragraphs.

- The distance matrix, which is used to calculate the affinity matrix can be either set directly using the method *setCostMatrix* or loaded from an ASCII-file using *loadCostMatrix.*

- *runClustering* performs spectral clustering by calling the necessary subroutines:

  - *eigenDec* This method executes steps 2 through 4 of the Spectral Clustering Algorithm 3.4.2. First the **DiagonalMatrix** D and the **Matrix** L are calculated from affinity matrix A. Then k eigenvectors are calculated from L using the *Jacobi* method from the `Newmat10` library. The method then calculates the renormalized rows of the eigenvector matrix and returns the normalized Matrix.
  - *initKM* initializes the data structures necessary for the k-means clustering. It copies the elements of the eigenvector matrix to the **KMdata** object, which is then used to create the centers for the k-means algorithm.

  - *executeAlgorithm* executes k-means clustering with the selected algorithm.
  - *getClustering* finally returns the assignments of the markers to clusters in an array, where the marker-ids can be used to index into the array for finding the associated cluster-id.

- *saveClusters* is finally used to store the grouped markers with their cluster ids in an ASCII file.

Table 4.15 in section 4.9 gives an overview over the parameters that influence the clustering process and can be set in *config.xml*.

4.4 Step 3: Computation of the Joint Positions

The part of the program this section is about can be considered the center piece of the system. It is responsible for the estimation of the joint positions. All previous steps basically are preprocessing steps for this component while the succeeding steps only refine the results. The calculations necessary for the computation of the joints are executed by two classes: The **JC** (short for joint calculator) and the **NLCG** (nonlinear conjugate gradient method) class.

JC implements the local and global minimization function as described in section 3.5 as well as some other methods necessary to process the clustered markers so they can be used in the functions. Furthermore **JC** contains the datastructures, which are necessary to ensure fast processing (described in more detail in 4.10). **NLCG** implements the nonlinear conjugate gradient method and is documented in section 4.5.

After the spectral clustering has grouped the markers, according to the limbs they belong to, a single instance of **JC** is created. This object then loads the marker data stored in a file as described in section 4.3. After an initialization phase a sample of frames is extracted from the marker data. This sample data together with the minimization function and some initialization values is then passed to an instance of **NLCG**. This is done for every possible combination of limbs. The minimization method then produces joint positions for the sample frames. These in turn are then refined by local optimization. The step, which has just been described, will be called the first stage in the following. Having done that, the minimization function can be used to evaluate the joints. This is done using the **MST** (Minimum Spanning Tree) class as described in 4.6. Therefore the limbs are viewed as nodes, while the joints are considered edges in a graph. The edge cost in this case is the return value of the minimization function. These cost values are passed to **MST**, which produces a skeleton structure by discarding the more expensive joints. The following calculations are considered the second stage of the joint calculation process. Now that the joints connecting limbs are identified the joint positions for every frame are calculated similarily to

int *k*	Number of clusters/limbs
int *numOfSampledFrames*	Quantity of frames, which have been selected for the first optimization phase
int *numOfFrames*	Number of tracked frames
int *numOfMarkers*	Overall number of markers attached to all limbs
iod::data_container *markers*	Contains (sampled) marker data ordered by frames and secondarily by marker id
iod::marker_container *perMarkerFrames*	Holds the markers ordered by marker ids and then by frames
iod::data_container *markersOriginal*	Contains all the marker data ordered by frames and secondarily by marker id
std::multimap <unsigned short, unsigned short> *clusterToId*	Used to find all the markers that belong to one cluster/limb
std::map <unsigned short, unsigned short> *idToCluster*	Finds the cluster id that belongs to a marker id
unsigned short *firstCluster, secondCluster*	ids of the two clusters involved in the calculation of the current joint
std::vector <iod::data_marker> *markersInFrames*	Buffer for the marker data for the current joint in a single frame used for communication between *localOptimization*() and *funcLocal*()
unsigned long *currentFrame*	id of *markersInFrames*
double *alphaAvg*	α in the minimization term
std::map <std::pair<unsigned short,unsigned short>, std::vector<double>> *av*	

Table 4.4: Member variables of **JC**

the sampled frames. Then the parameters of the skeleton are calculated and the joint positions further refined. For more details on that please refer to section 4.7.

The *member variables* of **JC** are listed in table 4.4.

JC is a singleton class instanciated in **MoCap**. The constructor of **JC** gets passed the filename and path of the output file from spectral clustering. It then does the necessary initializations of the datastructures and loads the parameters from *config.xml*. Other important methods are itemized in the following list.

- *loadClusteredMarkers* loads the positions, ids and corresponding cluster ids from the file it gets from the constructor and fills the datastructures. This way *k, numberOfMarkers, clusterToId, idToCluster* and *markersOriginal* are initialized to the correct values.
- *sampleFrames,* when executed by the constructor, copies samples of *markersOriginal* to *markers* and *perMarkerFrames.* The number of samples taken depends on the *SAMPLING* parameter in the *config.xml.*
- *initIdtoInt* is the next method to be called by the constructor and initializes *numberOfSampledFrames.*
- *calculateJointPos* is called by the **MoCap** class and gets passed path and name of the output file. It calls all the methods necessary to do the processing in the stage of the program, which includes parts of the initialization, calculation of the joints and finally creation of the XML elements for the output. *calculateJointPos* has a local array of doubles *x,* which has *numberOfSampledFrames**3 entries for the first stage. This array holds the position of the current joint over all (sampled) frames. These joint positions in turn are the unknowns we want to determine using the NLCG method. Since NLCG depends on good initialization values for good results the array is preset in the *initJointPos* method. After that x together with two variables for statistical evaluation and the two pointers to the methods *funcMain* and *dFuncMain* are passed to the **NLCG::***frprmn* method. For detailed information on this method please refer to section 4.5. After this global optimization the local optimization, as described in 3.5.2 is started. For that reason *av* has to be filled with values. This is done using the *calcAvLocOpt* method. Then *localOptimization* is executed with x as parameter. Then an **MST** object is created and handed a matrix containing the joint costs. That **MST** object produces a `stl::multimap` containing pairs of limbs that are supposed to have joints inbetween them. Thereafter the datastructures *markers* and *perMarkerFrames* are cleared and refilled with data from all frames. Also a **Skeleton** instance is created. After that the second stage of the optimization process is started, iterating only over the list of joints returned by **MST** instead of all possible joints as in the first stage. For every joint the positions as well as some statistical evaluation are set in the **Skeleton** object and stored in a `QDomElement`.
- *initJointPos* initializes the joint positions for the first stage before processing in the **NLCG** class. For the initialization some assumptions have to be made:
  - a joint is approximately half way between the centers of two limbs, when the joint is streched.
  - the markers are approximately evenly distributed around those centers.

– extreme positions, such as the knee completly bent are rare and do not last very long.

From these assumptions we can expect the markers' average position of the two limbs adjacent to the joint to be a good initialization.

- *calcAvLocOpt* calculates the average distance between the current joint and the adjacent limbs' markers. The results are stored in the *av* **stl::map.**
- *localOptimization* implements the idea described in 3.5.2. It uses the fact that in the ideal case the joint always stays at the same distance to a marker. It is assumed that the average distance over all frames computed by *calcAvLocOpt* is a good approximation to that optimal range. Therefore the joint position is optimized - again using NLCG - to be as close to the average distance to all markers as possible. This is done by minimizing the *funcLocal* function with the derivative *dfuncLocal*. Since the starting point is crucial to the result of the NLCG method *localOptimization* can be parameterized using MAXFRET and MAXFALLBACK. In case the return value of *funcLocal* after the optimization is bigger than MAXFRET the local optimization can be repeated with an old joint position as start value.

Now again its getting a little more mathematical while discussing the implementation of the formulas described and derived in section 3.5. I will start with the terms for the global optimization and continue with the local functions.

- *funcMain* implements the formula
  $jc(a,b) = \frac{1}{|b_a|+|b_b|} \sum_{m_n \in b_a \cup m_n \in b_b} \sigma(c, m_n) + \left[\alpha \cdot \bar{d}(c, m_n)\right].$

  The joint cost is the sum of all the involved markers's variances divided by the number of markers. It can be calculated for any combination of two limbs a and b whether or not there really is a joint between them. The above term is evaluated using some helper functions. They will be described in the following paragraphs in the order they are used in *funcMain()*.The processing is started by iterating over all markers m_n of limb b_a and b_b. Thereby the average distance between a marker and joint $\bar{d}(c, m_n)$ is calculated(c here is the joint position, which is contained in the array x in our case). The average is computed in the *averageDist* method. As mentioned before α is a member variable of **JC** and loaded from *config.xml*. The term between the square brackets is optional and ensures that the ball joint is close to the markers in case of an (approximated) rotation axis. $\sigma(c, m_n)$ is calculated using the *sigma* method and uses the results from *averageDist*. The results are summed ($\sum_{m_n \in b_a \cup m_n \in b_b}$) up during the iteration and then devided by the number of markers on the limbs ($|b_a| + |b_b|$) adjacent to the joint.

- *dfuncMain* calculates the derivative of *funcMain* for every unknown with the current value given by *x[]* and store it in *df[].* Note that every element of x is used to differentiate by. Thus the derivative has to be calculated (*numberOfFrames*3)* times, which makes the performance of this method rather important. The derivative can be mathematically denoted with the followin formula:$jc'_{xc_f}(a,b) = \frac{1}{|b_a|+|b_b|} \sum_{m_n \in b_a \cup m_n \in b_b} \sigma'_{xc_f}(c,m_n) + \left[\alpha \cdot \bar{d}'_{xc_f}(c,m_n)\right]$

  The derivative of the average distance $\bar{d}'_{xc_f}(c,m_n)$ is calculated in the method *derivAverage* and $\sigma'_{xc_f}(c,m_n)$ by *derivSigma. dfuncMain* also uses the *average* method, which is called once for every marker and passed to the other subroutines for reasons of efficiency. The derivative is first summed up over all frames for each marker and then all the results of the markers are added. The array *df* finally becomes a vector of numbers that points up the slope of the multidimensional function *funcMain* at point *x[].* The opposite direcition of that slope should then take as closer to the (at least local) minimum.

- *averageDist* calculates the average distance between the marker given by its id and the current joint positions for all (sampled) frames.

- *derivAverageDist* calculate the derivative by *x[startIndex]..x[startIndex+2]* of the *averageDist* function for one marker and stores them in a three element array.

- *sigma* computes the standard deviation of the distance between the marker given by its id and the current joint positions for all (sampled) frames

- *derivSigma* calculates the derivative by *x[startIndex]..x[startIndex+2]* of the sigma func given a marker and the derivatives of the average *da[]* and stores them in a threedimensional array *ds[].*

- *funcLocal* computes the difference between the distance of a marker and a joint in a specified frame and the average distance of these two entities.

- *dfuncLocal* produces the derivative of the function described in *funcLocal.* The values of *x[]* are then entered and the resulting three values returned in an array.

- *getStatsXMLElement* returns a `QDomElement` containing the evalutation of the joint cost minimization process. A joint is identified by the ids of the adjacent bones. The XML element holds information about the distances between the markers on the bones and the joint, the average distance between the marker and the joint over all frames and the standarddeviation thereof. Also contained are number of iterations necessary during the minimization to reach the specified threshold, the value of the minimization

int *numOfUnknowns*	Number of the unknown variables, the function value of which is to be minimized
double ***pcom,****xicom*	Dynamic arrays, which are initialized to size *numOfUnknowns.* They are used to simulate a one-dimensional function from a multidimensional one, a vector and a start point. *pcom* here is the start point and the vector *xicom* is the direction along which the functions variable is allowed to change.

Table 4.5: Member variables of **NLCG**

function at the result (below 1 is usually a good one) and the α of the minimization function.

- *saveData* writes a `QDomDocument` containing all the skeleton and animation data to a specified file. For a detailed documentation of the contents of a file please refer to section 4.13.

- some other methods exist for **JC**, which are not described here because they are not used in the current release but still might be useful for further experimenting.

4.5 Implementation of the Nonlinear Conjugate Gradient Method

NLCG implements the nonlinear conjugate gradient method described in section 3.6 and in many cases relies on the implementation remarks and code from [PTVF92]. Therefore at this point the methods used are not documented as in-depth as one might expect, since the interested reader can refer to [PTVF92]. The **NLCG** class can be used for any nonlinear optimization problem as long as a minimization function and its derivative are available. Vice versa any nonlinear optimization method can be used to minimize the joint cost function of section 4.4, as long as it implements the same interface as **NLCG.** The *member variables* of **NLCG** are documented in table 4.5.

The constructor of **NLCG** is passed the number of unknowns as parameter from which *numOfUnknowns, pcom* and *xicom* are initialized. Also *setParameters()* is called here. After this initialization the function *frprmn* can be called, which starts the minimization and is described with its helper methods in the following paragraphs.

- *frprmn* does the central processing in **NLCG**. It is passed multiple parameters.
  - *p[]* is an array of **double** which is used to hold the values of the unknowns (i.e. the coordinates of the joint positions in our case)
  - *iter* is used to return the number of iterations, which are needed to find the minimum.
  - *fret* holds the return value of the function after the minimization.
  - *func()* is a pointer to the function, which has to be minimized.
  - *dfunc()* is a pointer to the derivative of the minimization function.

  At first *func()* and its derivative are evaluated using the initialization values from *p*. After that the arrays with the residual, the search direction and the derivative can be set. Thereafter the iterative process of the minimization is started (Maximal *ITMAX* iterations). Additionally to the nonlinear conjugate gradient method I have implemented a random component, which is intended to escape a local minimum. This random vector is generated anew for every iteration using *RAND_MAX* and *MAXJITTER* and gets smaller linearly with the number of iterations. Then the method *dlinmin* is used to find the minimum in the current searchdirection. In case the desired minimum, which is specified by *FTOL* and *EPSBREAK*, is reached the *frprmn* exits and the current minimum returned to **JC.** If the criterions are not matched the new position is used to run *func* and *dfunc* again. Then the gradient and the search direction are calculated. Two choices for the latter can be specified in *config.xml* using *METHOD*. As mentioned in section 3.6 Fletcher-Reeves converges relatively sure for good initializations, while Polak-Ribiere is faster but sometimes gets stuck. After that the next iteration is started.

- *dlinmin* gets passed a point *p,* a direction vector *xi,* a dimension *n*, a variable to return the minimum value as well as pointers to *func()* and *dfunc()*. The task of *dlinmin* is to find the point on the vector *xi* starting at *p* where *func()* takes on a minimum. First the minimum has to be bracketed using the method *mnbrak*. Before the method is called the values of *p* and *xi* have to be copied to the *member variables pcom* and *xicom*, which are used for communication between the functions. The resulting bracketing interval is then passed to *dbrent*, which executes Brent's method to find the minimum. The resulting minimum is copied to *p* and *xi* and returned.

- *mnbrak* receives an initial guess for the bracketing as well as *func* and *dfunc* as parameters. Following the downhill direction from the initialization values new brackets are calculated by fitting a parabola and evaluating its minimum. For the evaluation the method *f1dim* is used. *TINY* is used for a return condition, which fires when no better bracketing can be reached.

The return value is a triplet of positions with function values, where the middle function value is smaller than the outer two.

- *dbrent* is passed the triplet from *mnbrak()* as well as a tolerance, *func()* and *dfunc()*. Then Brent's method, which is modified to use derivatives (*df1dim*) of *f1dim*, is executed. Break condition is the tolerance parameter, which holds the precision that has to be met. As a result the minimum value is returned and the middle value of the bracketing triplet holds the position along the direction vector calculated in *frprmn*.

- *f1dim* is an artificial one-dimensional function that uses the *member variables pcom*, *xicom* and *ncom* to emulate a one dimensional function for the minimization algorithms implemented in *mnbrak* and *dbrent*. Calls *func* and returns its value at position *pcom+x*xicom* (Note that *pcom* and *xicom* are the vectors used for communication, x is a scalar that is passed to *f1dim()* as parameter).

- *df1dim* is the derivative of *f1dim* and is similarly parameterized to it. The return value is the derivative at position *pcom+x*xicom*.

- *setParameters* loads the parameters from *config.xml*.

4.6 Minimum Spanning Tree

The class **MST** implements the algorithm of Prim as described in 3.7.7. The nodes of the graph are represented by the **Cluster** class. In addition to the data needed for Prim's algorithm **Cluster** objects contain information on the connections to other nodes. Therefore it is possible to travel within the graph, which is needed for the reorganization after the minimum spanning tree has been calculated. The edges of the spanning tree are stored in **MST** using a std::multimap as a std::pair node indices, while the nodes are stored as in a std::vector.

Cluster objects contain the information necessary for Prim's algorithm (priority value and their status e.g. fringe). Also information on where they are in the trees hierarchy (parent and children nodes) is stored. The **Cluster** class is derived from iod::Iteratee (4.10), which allows to retrieve an Iterator that can be used to access the children of a **Cluster** object.

The constructor of **MST** takes a **Matrix** object as input that contains the Adjacency matrix and thus the edge cost. The following gives a short description of the crucial methods in **MST**:

- *calculateMST* executes the algorithm. To change the algorithm - to for example Kruskal instead of Prim - one has to alter this method including the helper-methods *updateWith*(step 2 of the algorithm) and *findMinCluster*(step 3 of the algorithm).

- *optimizeMST* finds the node in the graph that has the most edges (i.e. the node that is assumed to be the most central) and makes it root. Then the structure information stored in the Cluster objects is rewritten accordingly.
- *saveMST* writes the tree to an ASCII file for inspection or further processing.
- *getEdges* returns the edges of the tree as std::multimap, which contains pairs of Cluster Ids. For each edge the Ids of the Cluster objects it connects is stored.
- *getRoot* returns the ID of the root node
- *getCluster* returns the Cluster specified by an index.

These methods provide two ways of accessing the spanning tree. On one hand *getEdges* can be used if only edge information is needed. On the other hand the minimum spanning tree can be traveled starting at the root, which reveals the hierarchical structure. This is important for the inverse kinematics implementation described in 4.7.2.

4.7 Step 4: Skeleton Representation and Parameterization

There are two main purposes of the classes described here. The first is to produce a skeleton with specified bone lengths. Secondly this skeleton has to be fitted back to the captured data. Therefore the rotations of all joints have to be calculated for all frames over the duration of the animation.

To perform the above tasks the classes **Skeleton, Limb, LimbFrame, Joint, JointFrame, Marker** and **PolySolver** are used. An instance of **Skeleton** is created before the joints are computed. It holds **Limb** and **Joint** instances. **Limb** in turn holds the positions of the markers attached to it, which are stored in **Marker** instances. Additionally the local structure of the skeleton (i.e. adjacency of limbs and joints) is recorded in **Limb** and **Joint**.

As a precondition the hierarchical structure of the skeleton has been determined by the class **MST** as described in section 4.6**.** In a first step the structure data has to be merged with the position data (joint- and marker-positions) to perform the next calculations. This is done while constructing the **Skeleton** instance. The next computations are rather similar to that described in [KOF05]. First, the lengths of the limbs are determined by averaging over all available frames (i.e. frames where enough markers are visible to the tracking system in order to perform the operations). Depending on the length of the captured time span a sample of frames can be used as well. To make the captured data comparable over the frames, the limbs have to be rotated and translated in a way that the marker and joint positions of two frames match as well as possible. [Hor87] offers

u_int16_t *id*	identifier of the marker
`std::vector <iod::data_markerPMF *>` *positions*	for each frame one instance of `iod::data_markerPMF` is created holding the position and if it was visible to the tracker (*exists*) or not

Table 4.6: Member variables of **Marker**

a method that does just that using quaternions. This algorithm is implemented in the classes **LimbFrame** and **PolySolver.** As a result the average marker- and joint-offsets relative to the centroid (the center of mass of the marker and joint positions) are produced.

These offsets are then used to reassemble a complete skeleton. Then the joint rotations are determined using inverse kinematics so the skeleton fits the data . This is done using the algorithm described in chapter 3.8. These results and some statistic evaluation are then written to an XML file using the `QT` library. The format used is a mixture of the `cal3d` XML format and the XML format used for tracking data of our optical tracker (see [PK07] for more details on that).

The classes are documented below in a bottom-up approach. The first class presented is the **Marker** class, because it is the smallest entity of the skeleton. It is followed by **Limb(Frame)**, **Joint(Frame)** and finally **Skeleton**. In addition the **PolySolver** class is documented here.

4.7.1 The Marker Class

The **Marker** class is the smallest class among those used for the skeleton reconstruction. It is a representation of a physical marker during the whole captured sequence. At the start of the algorithm it is initialized with the tracked positions of the markers and the id of the markers as they are found by the clustering algorithm as described in 4.3.

The *member variables* of **Marker** are listed in table 4.6.

The methods of **Marker** can be used to set and retrieve this data.

- *getPosition* returns a `iod::data_markerPMF` specified by the frame index.
- *getPositionRelativeTo* does the same as *getPosition* except it passes back the position relative to a **Vec3** passed as argument.
- *setData* sets ALL the positions of **Marker** to a vector passed as argument.
- *setId* sets the id of the **Marker.**
- *getId* hands back the id.

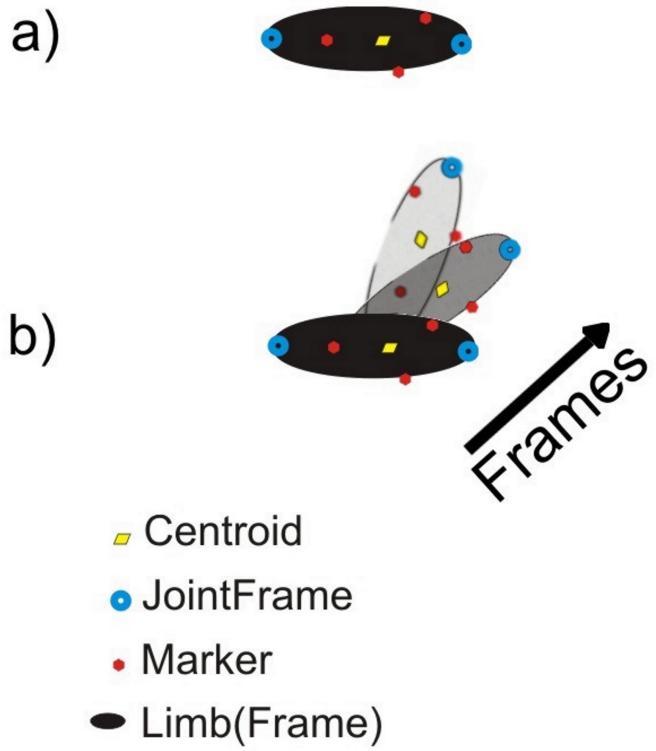

Figure 4.1: a) Shows a single **LimbFrame** with two attached **JointFrames, Markers** and the centroid. b) A Limb consisting of multiple LimbFrames.

4.7.2 The Limb And LimbFrame Classes

Limb and **LimbFrame** are probably the two most important classes of this part of the program. While **Limb** represents one bone of the skeleton at all frames, it contains a vector of **LimbFrame** instances, that hold the data of a bone in one specific frame. This can be seen in figure 4.1 a) and b).

The *member variables* of **Limb** are listed in table 4.7.

Note: A limb can have multiple outer joints, but only one inner joint!

The constructor of **Limb** takes an id and the number of frames as input. Many other methods exist for the **Limb** class, which for this reason I will group into three categories in order to provide concise view of the class. The first area that will be discussed is that of simple methods setting or retrieving data elements or performing basic operations. While the second section will deal with the more elaborate functions, the third with the interface to other classes. (this doesn't include **LimbFrame** which is defined to be a friend class of **Limb** and therefore has access to other methods as well)

u_int16_t *clusterId*	id of the Limb/corresponding cluster
u_int64_t *referenceFrame*	index of the first frame that contains all markers of the limb
Limb **parent*	adjacent limb that is higher in skeleton hierarchy
`std::vector` <LimbFrame*> *limbFrames*	for each frame an instance of **JointFrame** exists
Joint **innerJoint*	pointer to the **Joint** closer to the root (if existent)
`std::vector` <**Joint***> *outerJoints*	adjacent joints farther away from the root
`std::vector` <**Marker***> *markers*	markers that are placed on the limb
`std::vector` <**Vec3 ***> *outerJointsOffsets*	average offsets of the outer joint relative to the centroid
Vec3* *innerJointOffset*	average offset of the inner joint to the centroid
`std::vector` <**Vec3 ***> *markerOffsets*	average offsets of the markers relative to the centroid

Table 4.7: Member variables of **Limb**

Simple methods:

- *setClusterId* sets the *clusterId* to the specified value.
- *getClusterId/getId* return the *clusterId*.
- *setReferenceFrame* sets the *referenceFrame* to the id of the frame that is passed as parameter.
- *getReferenceFrame* hands back the id *referenceFrame.*
- *getRefFrame* returns a pointer to the **LimbFrame** instance that is identified by *referenceFrame.*
- *isMarkerOfLimb* checks if the marker determined by the id is placed on the limb.
- *getName* hands back the id of the limb converted into a `std::string`.
- *getParent* passes back a pointer to the **Limb** that is one level higher in the skeleton's hierarchy. Note that only the relationship towards the root can be referenced directly. In the other direction the adjacent limbs can only be referenced via the joints.

- *setParent* sets a **Limb** to be the *parent*.
- *getMarker* returns the **Marker** specified by an index. The parameter is not the global index of the **Marker** instance as it was determined by the clustering algorithm. It is only a local index with a range between 0 and the number of markers placed on the limb.
- *getMarkerPosition* passes back the position of a specified **Marker** in a certain frame. For this methods the same constraints as for *getMarker* apply.
- *getAvgPosMarker* returns the average position relative to the centroid of a specified **Marker** over all frames. The position is calculated after the limb has been transformed into the location and orientation of the reference frame.
- *getNumMarkers* hands back the number of the markers, which are placed on the limb.
- *getMarkerOffset* returns the offset of a specified marker relative to the centroid. The offset is calculated using position and orientation of the reference frame. Note that *calculateOffsets* has to be called before useful offsets can be retrieved.
- *getAvgPosInnerJoint* passes back the offset of the inner joint relative to the centroid.
- *setInnerJoint* sets a **Joint** to be the *innerJoint*, thus the joint that links the limb to either the root or a limb closer to the root than the current limb.
- *getInnerJoint* returns a pointer to the **Joint** *innerJoint*.
- *getInnerJointOffset* hands back the offset of the *innerJoint* relative to the centroid as it was calculated in *calculateOffsets*.
- get*JointFrame* hands back a pointer to the **JointFrame** specified by the id.
- *getAvgPosOuterJoint* passes back the average offset position of one joint over all frames for an outer joint specified by its id.
- *getAvgPosOuterJointsInFrame* returns the average offset position of the outer joints in a certain frame specified by the id.
- *getNumOuterJoints* passes back the number of joints attached to the limb minus one for the inner joint (i.e. the size of *outerJoints*)
- *addOuterJoint* adds a pointer to a **Joint** to *outerJoints*.

- *getOuterJointOffset* hands back the offset of a certain outer joint specified by the position in the `std::vector` *outerJoints.*
- *getOuterJoint* returns a pointer to a **Joint** specified by its id in *outerJoints.*
- *addMarker* adds a pointer to a **Marker** to *markers.*
- *getOuterJointFrame* returns a pointer to a **JointFrame** specified by the id of the joint in *outerJoints* and the frame id.
- *hasOuterJoint* passes back a boolean determining wether the **Limb** has limbs/joints attached to it, other than the inner limb/joint.
- *getLimbFrame* hands back a **LimbFrame** determined by the frame id.
- *setNumberOfFrames* sets *numberOfFrames.*
- *getNumberOfFrames* passes back *numberOfFrames.*

The more elaborate functions are:

- *calculateCentroids* iterates over all **LimbFrame**s and executes the methods of **LimbFrame** that are necessary in order to calculate the positions of the centroids (i.e. the centers of mass of marker- and joint-positions for all frames).
- *calculateMatrices* calculates the transformation matrices that rotate and translate the limb so that it matches the reference frame as well as possible. The reference frame is the first frame that contains all markers of the limb (i.e. is valid for our purpose).
- *calculateOffsets* calculates and stores the average offsets of the markers, outer joints and the inner joint (i.e. *markerOffsets, outerJointsOffsets* and *innerJointOffset*)

Finally there are methods to retrieve the results produced by **Limb.**

- *getAnimationXMLElement* returns a `QDomElement` holding the id of the limb, the number of keyframes (i.e. number of all frames in our case) and the rotation and translation of the inner joint for each frame.
- *getSkeletonXMLElement* is similar to *getAnimationXMLElement,* but returns only the rotation and translation for the reference frame. In addition, however, the rotation and translation, which is needed to transform a point from model space into bone space, is returned.

Limb **limb*	pointer to the **Limb** that the **LimbFrame** belongs to
u_int64_t *frameIndex*	position that the **LimbFrame** is located in the *limbFrames* vector of **Limb**
Vec3* *centroid*	center of mass of the markers and joints attached to the limb
bool *valid*	is true if all markers have been tracked in this frame
Quaternion* *rotation*	quaternion that can be used to rotate the limb from the reference frame into the current frame

Table 4.8: Member variables of **LimbFrame**

The **LimbFrame** class is closely tied to **Limb,** since much of the **Limb** data is stored as a `std::vector` of **LimbFrame**s. Every instance of **LimbFrame** contains data for one specific limb in a certain frame.

The *member variables* of **LimbFrame** are itemized in table 4.8

The constructor of **LimbFrame** takes the frame index and a pointer to the corresponding **Limb** as parameters and resets all other member variables. The other methods are similarly to the documentation of **Limb** divided into three subsections.

First of all the basic methods:

- *setLimb* sets the corresponding *limb*.
- *getLimb* passes back a pointer to the *limb*.
- *setFrameIndex* sets the *frameIndex* to the index of the `std::vector` *limbFrames* in the **Limb** instance.
- *getFrameIndex* returns the *frameIndex.*
- *setCentroid* sets the position of the *centroid*, which is computed by *calculateCentroids.*
- *setValid* sets the *valid* flag in case all Markers are visible to the tracker in the current frame.
- *isValid* hands back the *valid* flag.
- *setRotation* sets the *rotation* **Quaternion** to the parameter, that is passed as an argument.
- *getRotation* passes back the *rotation* **Quaternion**.

- *getGlobalRotation* returns a **Quaternion**, which can be used to rotate a vertex from modelspace to the bonespace of the **LimbFrame.**
- *getInnerJointFrame* hands back a pointer to the **JointFrame** that is adjacent to the *limb* and closest to the root.
- *getOuterJointFrame* passes back a pointer to the **JointFrame** specified by an index.
- *getInnerLimbFrame* returns a pointer to the **LimbFrame** closer to the root (if one exists).
- *getMarkerPosition* hands back the position of a **Marker** in *markers* determined by its index.
- *getPosRelCent* same as *getMarkerPosition,* except it calculates the position relative to the *centroid*.
- *getPosRelCentIJ* same as *getMarkerPosition,* except it calculates the position relative to the inner joint.
- *getPosRelCentTrans* same as *getMarkerPosition,* except it calculates the position relative to the *centroid*.
- *getPosRelCentOJ* same as *getPosRelCentTrans,* except it calculates the position relative to an outer joint specified by its index.
- *getPosRelCentIJTrans* similar to *getPosRelCentIJ,* but returns a copy of the marker position, which is transformed from modelspace into bonespace.
- *getPosRelCentOJTrans* same as *getPosRelCentIJTrans,* except it calculates the position relative to an outer joint specified by its index.
- *getAvgPosOuterJoints* passes back the average offset position of the **JointFrames** in *outerJoints*.
- *getNumMarkers* returns the number of markers which are visible to the tracker in the current frame.
- *getNumOuterJoints* hands back the size of *outerJoints.*
- *getMarkerOffset* same as in **Limb.**
- *getInnerJointOffset* same as in **Limb.**
- *getOuterJointOffset* same as in **Limb.**
- *setOuterJoint* sets the position of a specified **JointFrame** in **Limb::** *outerJoints* to a specified value.

- *hasOuterJointFrame* checks if the *limb* has an outer joint.
- *getRelTransRefFrame* passes back a vector containing the translation between the limb in the reference frame and the current frame.
- *getWorldTransMatrix* passes back a translation matrix that can be used to transform a vertex from modelspace into bonespace.
- *getInnerWorldTransMatrix* does the same as *getWorldTransMatrix,* except it returns the matrix for the bonespace of the inner **LimbFrame.**
- *getGlobalTranslation* returns a vector with the translation between the centroid of the *limb* in the reference frame and in the current frame.
- *getId* passes back the id of the *limb.*
- *getNumOfChilds* hands back the size of **Limb::** *outerJoints.*
- *getName* returns the id of *limb* as `std::string`.

The three more complex methods contained in **LimbFrame** are *calculateCentroid, calculateMatrix* and *calculateMatrixNew.* These are - among others - used for the inverse kinematics part of the program (see 2.6 for more details on that matter).

- *calculateCentroid* finds the center of mass of either all markers on the limb for the current frame, or all markers and the joints attached to the Limb. This is dependent on the Parameter *USE_JOINTS_FOR_CENTROID* in the *config.xml.* From the results of the first tests it can be said that in case of enough visible markers (three or more per limb) the joints should not be used.
- *calculateMatrix* produces a quaternion and a vector, which can transform the limb in the current frame in a way that it matches the reference frame as good as possible. This is necessary in order to calculate the limb lengths and marker offsets in **Limb** using the algorithm presented in 3.8**.** The translation is simply the vector between the centroids of the two **LimbFrames.** (Note that only **LimbFrames** where all markers are visible(*valid)* are used. So occluded markers cannot result in deviation.) For the calculation of the rotation quaternion the method described in 3.7 is used. A closed form algorithm introduced by Horn in [Hor87] is used to find the transformation that best matches two coordinate systems based on the measurement of some points in both systems. As already mentioned in 3.7 these points in our case are the markers and - in case there are less then two markers per limb - the joint positions. Following the algorithm the matrices are calculated and then passed to the **PolySolver** class, where the rotation

Limb* *innerLimb*	pointer to the limb closer to the root
Limb* *outerLimb*	pointer to the limb farther away from the root
stl::vector <**JointFrame***> *jointFrames*	for each frame exists an instance of **JointFrame**

Table 4.9: Member variables of **Joint**

quaternion is produced. This quaternion is used to set the *rotation* of the current limb.

- *calculateMatrixNew* calculates the rotation of the limb in the current frame for the inverse kinematics algorithm described in 3.8.2**.** This method works similarly to *calculateMatrix* except that for limbs, other than the root, the center of rotation is the inner joint instead of the centroid. Furthermore the joint positions are also used for the calculation of the matrix, since in cases, where markers are occluded there might not be enough measurements otherwise.

The results of the skeleton fitting algorithm can then be retrieved using the following two methods.

- *getAnimationXMLElement* passes back a `QDomElement` with the *frameId,* an estimate of the time that has passed since the start of the animation and rotation and translation relative to the parent bone. (*config.xml* holds the parameter FRAMES_PER_SEC, which can be used to adapt to different capture rates.)
- *getLimbFrameXMLElement* returns a `QDomElement` holding **Limb**:: *clusterId*, the id of the parent limb as well as translational and rotational information as calculated by *getRelTranslationParBone, getRelRotationParBone, getGlobalTranslation* and *getGlobalRotation.*

4.7.3 The Joint And JointFrame Classes

The **Joint** class is used to store and manipulate joint positions for all captured frames as they are calculated by the optimization algorithm as described in 4.5. A joint always has to be adjacent to exactly two limbs and between two limbs there can be only one joint. Thus a joint can be exactly defined by two Limbs. The *member variables* of **Joint** are listed in Table4.9.

The constructor of **Joint** is passed two **Limb** instances as parameters and can therefore be uniquely identified. Without data, however, an instance cannot be properly used, which is why the following method is used.

- *empty* returns true if no data has been entered in the instance

To set the position data that is handled down from the joint calculator (**JC**) to **Skeleton** and then finally to **Joint** another method is available:

- *setData* takes an array of *doubles* as input, which holds the positions of the specified joint over all frames.

Other important methods are listed in the following.

- *getJointFrame* passes back the **JointFrame** specified by its frame id. The ids have a range from 0 to (*numberOfFrames* -1).
- *getPosition* returns a **Vec3** containing the position of the joint in a certain frame. Parameterization is similar to *getJointFrame*.
- *getPositionRelativeTo* works analog to *getPosition* except it passes back the position relative to a parameter position. This is especially useful for local calculations, where positions relative to the centroid of a limb are of interest.
- *setPosition* sets the position of the joint in a specified frame.
- *getInnerLimb* hands back the **Limb** adjacent to the joint, which is closer to the root.
- *setInnerLimb* sets *innerLimb* to an actual instance.
- *getOuterLimb* returns the **Limb**, which is farther away from the root.
- *setOuterLimb* sets the *outerLimb* pointer to a specified instance of **Limb**.
- *getInnerJoint* checks if there is a joint adjacent to the inner limb, which is higher in the skeleton hierarchy and returns it.
- *addJointFrame* adds a frame with the specified position and index of the joint.
- *getInnerWorldTransMatrix* calculates a transformation matrix that transforms a point from modelspace into bonespace of the inner limb/bone for a specified frame (i.e. given the offset from the centroid this matrix can be used to determine the actual position of a marker in an arbitrary frame).
- *getLengthInnerLimb* passes back the bone length of the inner limb.
- *getXMLElement* returns a `QDomElement` containing positions for all frames.

Joint* *joint*	pointer to the **Joint** the **JointFrame** belongs to
Vec3* *position*	actual position of the joint at the frame with *frameIndex*
u_int64_t *frameIndex*	index of the frame in the *jointFrames* vector in the corresponding **Joint**

Table 4.10: Member variables of **JointFrame**

The **JointFrame** class is responsible for storage and manipulation of a joint position in one specific frame. The *member variables* are itemized in table 4.10.

JointFrame is constructed with an index, the 3D position and a pointer of the **Joint** it belongs to. The following methods are available to edit the data.

- *getName* hands back the *frameIndex* converted into a `std::string`.
- *getPosition* passes back the *position.*
- *getPositionRelativeTo* returns the *position* relative to a specified vector.
- *setPosition* sets the position to the specified vector.
- *getVecToInnerJoint* passes back the vector to the joint that is adjacent to the inner limb and higher in the hierarchy of the skeleton if it is available.
- *getJoint* returns the **Joint** that **JointFrame** belongs to.
- *setJoint* sets the **Joint** that **JointFrame** belongs to.
- *getXMLElement* passes back a `QDomElement` containing *position* and *frameIndex.*

4.7.4 The Skeleton Class

Skeleton is the central class for this part of the system. It provides the interface to other components of the program and holds pointers to all **Limb, Joint** and **Marker** instances. These are stored in the form of an `stl::vector(`e.g. *stl::vector<Limb *>limbs).* Figure 4.2 shows a schematic view of the skeleon structure. Furthermore table 4.11 shows the *member variables* of **Skeleton**.

The constructor of **Skeleton** takes as arguments a reference to an **MST** object (section 4.6), a `marker_container(`see section 4.10 for details) and a `stl::map<unsigned short,unsigned short>` (maps the id of the marker to the id of the limb it is placed on).

The latter two are used in the constructor to create the **Marker** instances and set the references to their corresponding limbs. The method

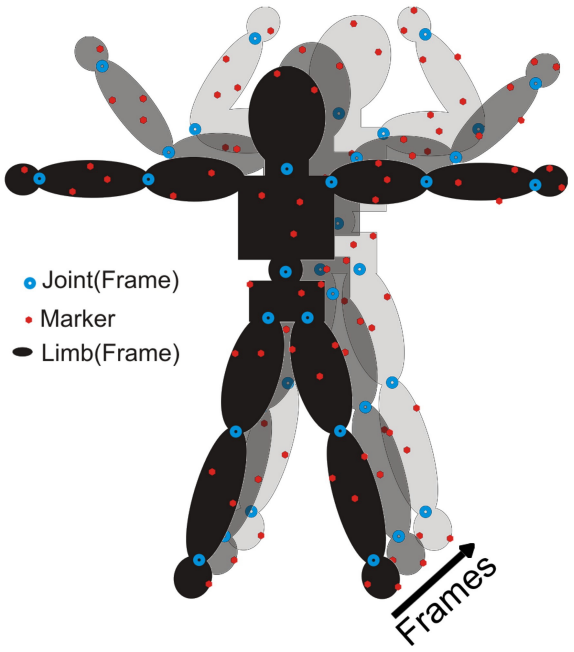

Figure 4.2: Schematic view of the **Skeleton** structure with **Joints** and **Markers**. Note that for each **Joint/Limb** there exists an instance of **JointFrame/LimbFrame** in every frame.

stl::vector<**Limb** *> *limbs*	limbs(or bones) of the skeleton
stl::vector<**Joint** *> *joints*	joints between the bones
stl::vector<**Markers***> *markers*	all the markers placed on the body
u_int64_t *numberOfFrames*	number of the tracked frames
Limb* *root*	pointer to the root of the skeleton

Table 4.11: Member variables of **Skeleton**

- *copySkelStructure* recursively copies the skeleton structure from the **MST** object to the **Limb** and **Joint** instances.

Other important private methods in **Skeleton** are listed in the following.

- *calculateSkeletonParameters* iterates over all limbs and calls limb member functions to calculate the parameters (limb lengths, offsets, rotations etc.)

- *findJoint* returns the joint, which is identified by the ids of the adjacent limbs.
- *addJoint* adds a joint to the skeleton.
- *getLimb* passes back the limb with the specified id.
- *addLimb* adds a limb to the skeleton.

Note: Some of the following functions' purposes is to create XML elements, which are then stored in a XML file. The structure of the output XML file and its elements can be found in 4.9. For documentation on the `Qt` library and its XML components please refer to [Tro07].

- *getSkeletonXMLElementInFrame* hands back a `QDomElement` which stores all the available information for a certain frame (i.e. rotations for the joints and translations for the bones, as well as the marker positions as they are calculated from the offsets and skeleton parameters)
- *getSkeletonXMLElement* returns a `QDomElement` containing the structure of the skeleton (i.e. joint positions, the offsets between joints, angels of the joints for a reference frame)
- *getAnimationXMLElement* passes back a `QDomElement` holding the animation data for all frames (joint rotations and translations/bone-lengths)
- *getAnimationXMLElementOfLimb* returns the animation data for a specific limb.
- *setNumberOfFrames, getNumberOfFrames* and *getNumberOfLimbs*
- *calculateSkeletonParameters* calls for each limb the methods that calculate the parameters (offsets, length, rotations etc.).
- *recursiveCalcMatricesNew,* starting with the root-limb, executes the methods that produce the joint rotations of the parameterized skeleton so it best fits the captured data.

The following methods provide the interface to other classes.

- *setDataJoint* sets the positions to the values given as parameter.
- *getJointsXMLElement* returns a `QDomElement` containing the positions of all joints over all frames.
- *getJointXMLElement* passes back a `QDomElement` with the positions of a specified joint over all frames.

double *A* **[5][5]**	is used to hold the input matrix and intermediate results during most of the processing
RUNTESTSVAR	This variable is loaded from *config.xml.* If true the eigenvectors are calculated using the `newmat` library instead. This can be useful since the closed form methods crash in some cases. Especially if the structure of the skeleton is wrong this tends to happen.

Table 4.12: Member variables of **PolySolver**

- *getSkelAniXMLElement* hands back a `QDomElement` holding all the skeleton and animation data.
- *getMarkerXMLElement* returns a `QDomElement` containing all the information on markers that has been collected during the previous processing steps. That includes the tracked positions, the bones they belong to as well as the averaged offsets from the centroids of the limbs. Furthermore the positions of the markers are reconstructed from the offsets and the joint rotations to provide a means for error measurement.

4.7.5 The PolySolver Class

PolySolver is the last class used to reconstruct the skeleton. It is used to execute steps 3 (or 4 depending on the mode **PolySolver** is run in) through 8 of the rotation finding algorithm described in 3.7. The three preprocessing steps are done in **LimbFrame::***calculateMatrix(New).* Thus as an input **PolySolver** has a 3-by-3 or 4-by-4 matrix constructed by the component products of the corresponding points in two coordinate systems. As an output **LimbFrame** delivers a rotation quaternion, which can be used to rotate one coordinate system so the measured points match the points in the other as well as possible. The *member variables* of **PolySolver** are listed in table 4.12.

The constructor of **PolySolver** gets passed an array of doubles, which is used to initialize the matrix *A,* as well as an array *B* for the return values and the *mode* flag. In case *mode* is 0, 16 values are supplied and the Matrix is set directly. If mode is 1 then 9 values are supplied which are used to calculate a symmetric matrix. After that, the eigenvector of *A* has to be found which has the largest eigenvalue. The eigenvector, which at the same time can be seen as quaternion representing the optimal rotation, is then stored in *B*. These calculations are performed using the following methods:

- *setMatrix* and *setMatrixByCompProduct* are used to set (mode 0) or calculate A (mode 1) from the array passed as argument.

- *calculatePoly* is the main method that calls the other methods necessary to produce the largest eigenvector of the matrix A. It uses the functions *calculateCoefficient, solveQuartic, echelon* and *calculateEigenvector* in this exact order.

- *calculateCoefficient* calculates the coefficients of the charcteristic polynomial of A. This method uses the formulas described in [Hor87] to produce the coefficients of the characteristic polynomial of matrix A and stores them in an array passed as parameter.

- *solveQuartic* can solve a general 4th order equation given by an array of coefficients *coeff*.

  $0 = coeff[0] + coeff[1] * x + coeff[2] * x^2 + coeff[3] * x^3 + coeff[4] * x^4$

  The results of the equation are returned using the array re and im for the real and imaginary part. Both are 4-dimensional. Note that for the use in our case only the real part is needed, since the equation we obtain from the matrix has real eigenvalues and eigenvectors. The equation is solved by reducing it to equations of smaller order following the basic idea of Ferrari. These are then solved by the methods *solveCubic, solveQuadric* and *solveLinear*. For the mathematical background please refer to [Vil04]. The formulas in my code have been taken from Olaf Müller's java code [Mül03].

- *solveCubic* solves a general 3rd order equation given by an array of coefficients *coeff*.

  $0 = coeff[0] + coeff[1] * x + coeff[2] * x^2 + coeff[3] * x^3$

- *solveQuadric* can solve a general 2nd order equation.

- *solveLinear* solves a linear equation.

- *echelon* transforms the A matrix into row echelon form. This means that

  1. the first non zero element in each row is 1 and
  2. all elements below are zero

  i.e. in the form $\begin{matrix} 1 & a & b & c \\ 0 & 1 & d & e \\ 0 & 0 & 1 & f \\ 0 & 0 & 0 & 1 \end{matrix}$

The methods *echelonRec, pivotSearch, swap* and *addRows*, which are described in the following, are all helper methods of *echelon*. For a more detailed mathematical description please refer to [Mey00] or another linear algebra book.

- *echelonRec* is called recursively. Needs two indices (row- and column index) as parameters, which are used to define the matrix's area of interest (i.e. the manipulated elements are those right/below the specified values).
- *pivotSearch* looks for the (absolute) largest element in a specified column of the A matrix. The search starts in a row, which is determined by a parameter, and goes down.
- *swap* exchanges the ith and the jth row of the matrix A to place a pivot element at the desired position. This process is called partial pivoting. Full pivoting would be swapping rows and columns.
- *addRows* adds the ith row multiplied by a certain factor to the jth row.
- *calculateEigenvector* produces the eigenvector of the largest eigenvalue of matrix A. Having calculated the eigenvalues and having transformed the matrix A into row echelon form, this method calculates the eigenvector corresponding to the largest eigenvalue of matrix A. The eigenvector is then normalized to unit length.
- *runTests* calculates the eigenvector using an iterative approach, when the *RUNTESTSVAR* flag is set.
- *printMatrix* prints matrix A to the screen.
- *initMatrix* initializes the elements of matrix A to 0.
- *cosh* passes back the cosinus hyperbolicus of the given value
- *sinh* returns the sinus hyperbolicus of the given value.

4.8 Helper Classes

In this section some of the helper classes used throughout the system are described. These includes classes for matrix calculations, vector representation and iterators.

4.8.1 Transformation Matrix

The class **Matrix4x4** represents a square transformation Matrix with 16 elements that can be used to transform a 3D vector. The vector has to be an object of the class **vec3** as described in the next section or a array of doubles. **Matrix4x4** provides a number of set-methods that can be used to set specific transformations (e.g. a rotation around the x-axis). Also standard matrix operations like multiplication, translation or inversion are available. The methods available are:

- Constructors taking the following elements as parameter:
  - a reference to a **Matrix4x4** object
  - a float array with 16 entries
  - a quaternion
- For setting the whole matrix or parts of it some methods are available
  - *set* takes a 16 element double array as an argument and sets the matrix accordingly
  - *setMatrixbyQuaternion* sets the matrix's elements to the rotation described by the quaternion that it has been passed as four doubles.
- *loadIdentity* resets the matrix to a $4x4$ identity matrix
- Matrix multiplications
  - *preMultiply* multiplies the current instance M of **Matrix4x4** with the matrix A it received as an argument and stores it in M. Thus $M = AM$
  - *postMultiply* multiplies the current instance M of **Matrix4x4** with the matrix A it received as an argument and stores it in M. Thus $M = MA$
  - *transformVec3* multiplies a vector v of dimension four, given as an argument, by the matrix: $v = vM$
  - *transformVec3Pre* multiplies a vector v of dimension four, given as an argument, by the matrix: $v = Mv$
  - *set(Inverse)Translation* sets the translation of the current matrix to the (inverse) values of the double array or **Vec3** object passed as an argument and erases any previous values.
  - *set(Inverse)RotationRadians/setRotationDegrees* sets the rotation part of the matrix to the (inverse) rotation given by an array of doubles that contains the Euler angles

4.8.2 3D Position/Vector

Vec3 is a class used to represent and manipulate a 3D vector. Therefore some basic algebraic operations are available as well as some specifically designed for our application. For the storage of the three coordinate values *member variables* of type **double** exist. The constructor can either be initialized with a **Vec3** reference, three **double** or an array of **double**. Also the standard constructor can be used. The other methods are itemized in the following list.

- *set* sets the coordinates to another **Vec3** passed as reference or to three single values.
- *reset* sets the coordinates to the point of origin.
- *getData* writes the coordinate values to an array, which has to be passed as an argument.
- The binary operators *+,-,^* and *%* return the sum/ difference/ cross product/ dot product of two **Vec3** objects.
- The operations * and / are used for scalars.
- *length* returns the length of the vector.
- *lengthsq* returns the sum of squares of the coordinates.
- *getDistTo* returns the distance to another **Vec3** object handed over as parameter.
- *normalize* shortens/stretches the vector to length 1.
- The operator [] returns the coordinate defined by an index. (e.g. 0 for x)
- *rotateX, rotateY* and *rotateZ* rotates the **Vec3** object around the X, Y or Z axis by an angle specified in radians.
- *componentProduct* calculates the component product of the elements of two **Vec3** and stores it in a 3 by 3 matrix passed as argument

4.9 The Parameter XML File and the Parameters Class

The parameter XML file named *config.xml* is used to store values, which influence the behavior of almost all parts of the motion capture system. Due to the fact that lots of parameters are included, the file is organized hierarchically. The root is the <**Parameters**> element. Its sub-elements group the parameters into categories depending on the part of the program they are used in. The parameter elements of the subsystems can be of the three types **double, int,** and **string** (Note that **int** is also used as **boolean**). Tables 4.13, 4.14, 4.15 and 4.16 list the parameters as they appear in the file. The XML elements are loaded into the program using the **Parameters** and the **Handler** class.

Name	Type	Description
tempCorFileName	**string**	Default name of the *.mfml file which is to be loaded
specCluFileName	**string**	Default name of the file which is used to store the marker data after the temporal correspondence over the frames has been established
jointFinderFileName	**string**	Default name of the file holding the clustered markers
skelFitFileName	**string**	Default name of the file, where the structure of the skeleton is stored
saveFileName	**string**	Default name of the XML file that holds the results and evaluation of the whole program

Table 4.13: Parameters of GUI in config.xml

4.9.1 The Handler Class

The `Qt` SAX parser is used to process the *config.xml.* This makes sense in our case since the parameters are loaded only once at the start of the program and we don't want to modify them nor is there a need to keep the whole tree in memory. In order to read a file using the SAX parser one has to implement a handler class, which inherits from the `QXmlDefaultHandler` class. This is done by the **Handler** class, which overwrites the *startElement* method so the behaviour can be customized, when the parser reads a new element. Apart from that function **Handler** has only one additional method and no *member varibles.* A short description of the mehtods can be found in the next paragraphs.

- *startElement* processes parameter elements. Each element needs to have two attributes - one for the value and the second for the type. Depending on the type of the parameter the **Parameters** singleton is called with the set method for either a numerical value or a string.
- *fatalError* is called in case an exception occurs during parsing and writes an error message to the standard output.

4.9.2 The Parameters Class

The **Parameters** class is a singleton, which means it has only one instance and a static *getInstance* method. This is convenient, because many classes need to

Name	Type	Description
SAMPLING	**int**	Determines how many frames are used to calculate the variations of distance between markers (Note that the matrix produced by this values is calculated in the **CorrespondenceFinder** class but used by the clustering algorithm)
MINRANGE	**int**	Marker sets which have less than *MINRANGE* frames are removed. This is done because tracking accuracy is usually rather bad if a marker pops up just for a view frames and then gets occluded again.
TRIMRANGE	**int**	The beginning and end of each Marker set is cut of by *TRIMRANGE* frames to ensure accuracy.
TIMEONEFRAME	**int**	Time between two frames (depending on the capture rate)
ADDRANDOMTC	**int**	This variable determines whether or not jitter is added to the input data. This can be useful for testing purposes, when clean data is generated by some animation program.
MAXJITTERTC	**double**	Maximum of added random value in millimeter.
SAMPLINGMSTC	**int**	Sampling interval for the frames that are picked to find corresponding markers.

Table 4.14: Parameters of Temporal Correspondence Finder in config.xml

load their parameters from this instance. **Parameters** also has some *member variables* as listed in table 4.17.

Additional methods of **Parameters** are listed below. Note that the constructor (standard constructor) is only called in *getInstance()*.

- *getParameterValue* receives a name and returns its numerical value.
- *setParameterValue* gets a name and a numerical value and sets the parameter accordingly in *values*.
- *getStringParameterValue* returns the string stored as value for the parameter with the name it gets passed as argument.

Name	Type	Description
SPECCLUEPS	**double**	Used to determine if a value is 0 or sufficiently close to assume it is not 0 due to numerical errors
SIGMAQUAD	**double**	Determines the value of the constant global sigma, which is used for calculating the affinity matrix
LOCALSIGMA	**int**	0...global sigma is used, 1... local sigma is used
M	**int**	Number of points used for the calculation of local sigma
ALG	**int**	Selection of different implementations of k-means (0-Lloyds, 1-Swap, 2-Hybrid, 3-LocalHybrid)
POLYEPS, SOLVEPS, TINYEPS	**double**	The different "eps"s are used in the methods *echelonRec, calculateEigenvector* and *calculatePoly* to determine if a value is 0 or sufficiently close to assume it is not 0 due to numerical errors
PI	**double**	Mathematical constant
ALPHA	**double**	Constant in the formula that is minimized in order to find the best joint positions
ALPHAWEIGHT	**double**	Constant in the formula that is minimized in order to to find the best joint positions
SAMPLING	**int**	Determines for how many frames the joints are calculated (computing joints for a sample of frames only is used to find good initialiization values for the actual processing step)
MAXFRET	**double**	Is used as a threshold for the local optimization
MAXFALLBACK	**int**	For the local optimization the position of the joint in a previous frame is used as initialization value. This parameter gives the number of frames the algorithm is is allowed to go back.

Table 4.15: Parameters of Spectral Clustering, JointFinder and JointFinder:: JC in config.xml

Name	Type	Description
FTOL	**double**	Is used together with *EPSBREAK* to determine when the minimization algorithm has reached its goal and should break. (normal termination)
EPSBREAK	**double**	Is needed when converging to exactly zero function value
ITMAX	**int**	Specifies the maximum number of iterations the algorithm is allowed to run. If this value is sufficiently large and nevertheless is exceeded the algorithm is most likely trapped in a local minimum.
EPS	**double**	Small value used by the minimization algorithm for a break condition.
GLIMIT	**double**	"Is the maximum magnification allowed for a parabolic-fit step" [PTVF92]
ITMAXDBRENT	**int**	Maximum number of iterations that Brent's method is allowed to take
ZEPS	**double**	Small value used by Brent's method for a break condition.
ADDRANDOM	**int**	Determines whether or not a random value is added to the joint position during the optimization process in a simulating annealing approach. In some special cases this helps escaping local minima, but also increases processing time.
MAXJITTER	**double**	Maximum of added distance in the first iteration.
TOLBRENT	**double**	Tolerance used by Brent's method and line minimization method.
METHOD	**int**	Chooses the method for calculation of the new search direction (0...Fletcher-Reeves, 1...Polak-Ribiere)

Table 4.16: Parameters of JointFinder:: NLCG in config.xml

- *setStringParameterValue* same as *setParameterValue* except the type of the parameter has to be a string and is set in *stringValues*.

- *clear* empties *values* and *stringValues*.

std::map <std::string, double> *values*	Maps the name of the parameter to the numerical value. For the sake of simplicity also **int** and **double** are stored in the same map.
std::map <std::string, std::string> *stringValues*	Maps the name of the parameter to the **string** value.
static Parameters ***instance**	The only instance of the **Parameters** class.

Table 4.17: Member variables of **Parameters**

4.10 Implementation Data Structures

During the different processing steps massive amounts of computations are performed using the captured data. Furthermore the markers are captured at up to 200 times per second, which results in rapidly growing amounts of 3D positions that have to be stored. It is therefore crucial to use data structures, which support fast access and at the same time contain only the minimum of the needed data to limit memory consumption. Therefore, instead of using for example one marker class for the whole program, multiple classes and structures exist and are specifically designed for each processing step. Most classes make use of the `stl` (standard template library), which makes some understanding of `std::map`, `std::vector` and `std::iterator` etc. helpful for comprehending the following. I will start with the smallest entity: the marker.

4.10.1 The Marker

The only attribute of a marker, which is needed in all processing steps is the position. Therefore all marker structures inherit from **Vec3**, which offers methods to store and manipulate a 3D position or vector (see section 4.8.2 for details on that class). The next **struct data_idmarker** adds an id to the marker, which is needed for the temporal correspondence (section 4.2). **data_idmarker** is then extended with a cluster id to **data_marker**. This structure is used for the spectral clustering algorithm and the joint finder (Sections 4.3 and 4.4). For skeleton parameterization finally the class **Marker** is used, which is described in section 4.7.1.

Depending on the intended use the markers are addressed in two ways. The first mode of access is to read/write a single marker over all frames then continuing with the next marker. The other option is addressing one marker in a frame after the other then jumping to the next frame. The structures for the latter mode have been described in the above paragraph. For the calculations of the joints, however, a structure like **data_markerPMF** (the PMF stands for "per marker frame") is needed to provide fast and simple access. Like the other marker structures

u_int32_t *n_markers*	Number of markers visible to the tracker.
u_int64_t *timestamp*	Time passed since the start of the animation as read from the *.mfml file.
std::vector <iod::data_marker> *markerVector*	Markers, which are visible to the tracker.

Table 4.18: Member variables of **dataframe**

it holds the position, but no id or cluster information. This is not needed since **data_markerPMF** is used only as element of an `std::vector`, where the id and cluster is stored once for a marker. For simplicity these `std::vectors` contain as much **data_markerPMF** entries as there are frames. This means that there even exists one if the physical marker is occluded to the tracker. Therefore a flag has to be set in **data_markerPMF** if the marker actually exists in the frame.

4.10.2 The Frame

For access frame by frame the class **data_frame** is being used. This class has some *member variables* listed in table 4.18.

data_frame can be constructed either using a `std::vector` of **data_markers** or as an empty frame. Furthermore **data_frame** has a number of methods to manipulate the data and compare two objects of type **data_frame**.

- *getTimstamp, setTimestamp, setNumOfMarkers* and *getNumOfMarkers* return/set the according member variables.
- *pushBack* adds a **data_marker** to the *markerVector.*
- *clear* empties the *markerVector.*
- *setMarkerId* sets the id of a marker at specified position in *markerVector* to a certain value.
- *globalTransform* transforms a `std::vector` containing pairs of positions (`std::pair`<**Vec3,Vec3**>) using two different matrices, which are passed as arguments.
- *getDist* returns the sum of distances between the markers of two different **data_frames** after they have been transformed using the *globalTransform* method. This is used when the temporal correspondence of two markers is looked for. See section 4.2 for more details on that matter.

u_int32_t *n_markers*	Number of succeeding f_marker blocks
u_int64_t *timestamp*	Time since the start of the capturing session

Table 4.19: Member variables of **f_sfml**

4.10.3 The Whole Data Set

For quick access the frames and markers are stored in `stl` containers by the classes which use them. To somewhat unify the use of these containers I created some **typedefs.** These are named *marker_container* and *data_container*. While the *marker_container* is intended to access a single marker in different frames, the *data_container* contains all the marker data frame by frame. Therefore the different classes for markers can be used as described in section 4.10.1.

The *marker_container* is defined as:

std::map<unsigned short, std::vector<data_markerPMF>>

Thus the index of the marker can be used to find a vector that contains the positions (or an invalid flag) for each frame. The *data_container* is defined as:

std::vector<data_frame> data_container

A vector of frames, where the index of the vector can be used to access an arbitrary frame. These two types make possible fast iterations over the markers/frames.

4.11 mfml File Format Description

The *.mfml file is the interface between the tracker server and the motion capture system as described in this thesis. Thus first the tracker (as described in section 2.5) has to record data. After processing by the tracker software 3D positions of the markers and some information about the reconstruction are available. This data is then written to an *.mfml file for further processing. The file format was designed by Thomas Pintaric, who is a member of the virtual reality research group at the IMS institute of the Vienna University of Technology. Unfortunately no documentation exists so far and I will therefore describe it here in short.

An *.mfml file is a binary file that can be saved/loaded using two **struct** and a **typedef.** These are **f_sfml** (frame), **f_marker** (marker) and **mfml_container** (whole data set). The latter is similarily defined to the **data_container** described in section 4.10 **(std::vector <std::pair <f_sfml, std::vector <f_marker> > >).** The difference between the two is that here the marker data is not included in the frame but paired together.

4.11.1 The f_sfml Struct

The **f_sfml** struct is used to store some information about a frame. It therefore has two *member variables* as described in table 4.19.

float *center*[**3**]	Coordinates
float *reproj_err*	Average reprojection error in millimeter
u_int32_t *n_blobs*	Number of blobs used for projective reconstruction
core::unique *id*	A unique id for every marker in every frame

Table 4.20: Member variables of **f_marker**

Two methods are available for **f_sfml** - *read_from_file* and *write_to_file,* which get passed a file pointer and read/write the two variables from/to file.

4.11.2 The f_marker Struct

The **f_marker** struct saves position and other data as it is generated by the tracker. The *member variables* are listed in the following.

As **f_sfml f_marker** has a *read_from_file* and *write_to_file* method and additionally can calculate the distance between two markers using *getDistTo(***f_marker** **&***secondMarker*).

4.12 The csm File

The csm format has been developed by Autodesk (the software producer developing 3d Studio Max). It is primarily used to store unconstrained joint positions without any information about the underlying skeleton. When used for animation the positions are then imported into the animation program. There they have to be manually connected to the model by the animator.

In our case the stored values are not the positions of the joints but those of the markers. They are then loaded into the program along with their ids. (which can be retrieved from the order the markers appear in the file)

For evaluation also the joint positions were exported so they could be compared to the findings of the algorithms.

4.13 The Output XML File

The output of the motion capture system is designed to be used for animation as well as evaluation of the quality of the motion capture data. Therefore not only the skeleton data is included but also some statistical values of the different processing steps. As hinted in section 4.7 the format is a mixture of the `cal3d` XML format and the XML format used for tracking data. The output is explained using a commented (and shortened) sample file. The use of "..." indicates that parts similar to others, which have already been explained have been removed from the file.

```
<StatsElements>      //evaluation of the joint cost minimization process
```

```
<Joint SecondBone="0" FirstBone="1"      /a joint is identified by the ids of the adjacent bones
    Alpha="0" >          //alpha of the minimization formula
<DistanceStats>          //information about the distances between the markers on the bones and the joint
<FirstBone>
<MarkerDistToJoint MarkerId="0"
    AverageDist="100"     //average distance between the marker and the joint over all frames
    Sigma="0.3"      //standard deviation of the distance
    />
<MarkerDistToJoint ...
</FirstBone>
<SecondBone>
<MarkerDistToJoint ...
</SecondBone>
<CalculationStats NumberOfIterations="24"      //number of iterations necessary during the minimization to
reach the specified threshold
    FuncValueAtSolution="0.3387904120855729" />     //value of the minimization function at the result (below
1 is usually a good one)
</Joint>
<Joint...
</StatsElements>
<SkeletonAndAnimation>     //contains the skeleton parameters and animation data
<SKELETON      //hierarchical skeleton structure similar to cal3d
    NUMBONES="3" >     //number of bones in the skeleton
<BONE PARENT="-1"      //parent "-1" means root, any other the id of the parent bone
    BONEID="1" >     //id of the bone
<vector length="3" name="TRANSLATION" >     //relative translation to parent bone (zero for root)
<vec_elem val="0" pos="0" />
<vec_elem val="0" pos="1" />
<vec_elem val="0" pos="2" />
</vector>
<vector length="4" name="ROTATION" >     //relative rotation to parent bone (quaternions are used to rep-
resent rotation)
<vec_elem val="-0.4394910733252021" pos="0" />
<vec_elem val="0.6901917616168408" pos="1" />
<vec_elem val="0.4056551345082463" pos="2" />
<vec_elem val="-0.4073411844029531" pos="3" />
</vector>
<vector length="3" name="LOCALTRANSLATION" >     //translation to bring a vertex from model space
into bone space
<vec_elem val="507.4327708016544" pos="0" />
<vec_elem val="90.47017325877077" pos="1" />
<vec_elem val="1096.061977017274" pos="2" />
</vector>
```

```
<vector length="4" name="LOCALROTATION" >     //rotation to bring a vertex from model space into bone space
<vec_elem val="-0.6901918053627014" pos="0" />
<vec_elem val="-0.4056551456451416" pos="1" />
<vec_elem val="0.4073411524295807" pos="2" />
<vec_elem val="0.4394910931587219" pos="3" />
</vector>
</BONE>
<BONE...
</SKELETON>
<ANIMATION>     //animation data is stored here similarily to cal3d
<TRACK NUMKEYFRAMES="554" BONEID="1" >     //every bone has its own track which in turn contains the keyframes (in our case every frame is a keyframe)
<KEYFRAME FRAMEID="0" TIME="0" >     //id of the frame and recorded time in sec (starts at 0 for the first frame)
<vector length="3" name="TRANSLATION" >     //relative translation to parent bone
<vec_elem val="0" pos="0" />
<vec_elem val="0" pos="1" />
<vec_elem val="0" pos="2" />
</vector>
<vector length="4" name="ROTATION" >     //relative rotation to parent bone
<vec_elem val="-0.4394910733252021" pos="0" />
<vec_elem val="0.6901917616168408" pos="1" />
<vec_elem val="0.4056551345082463" pos="2" />
<vec_elem val="-0.4073411844029531" pos="3" />
</vector>
</KEYFRAME>
<KEYFRAME ...
</TRACK>
</ANIMATION>

<RawJointPositions>     //output of the joint cost minimization algorithm (no parameterized skeleton yet, distances between joints can vary)
<joint1-0>     //joint identified by bone ids
<JointFrame FrameIndex="0" >
<vector length="3" name="0" >     //position of the joint in the frame
<vec_elem val="679.8912338402603" pos="0" />
<vec_elem val="73.63353316984644" pos="1" />
<vec_elem val="1078.880517499108" pos="2" />
</vector>
</JointFrame>
<JointFrame...
</joint1-0>
<joint...
```

```
</RawJointPositions>
<FinalJointPositions>     //joint positions after the skeleton has been parameterized and fitted back to the data
<JointPositions>
<joint1-2>
<JointFrame FrameIndex="0" >
<vector length="3" name="0" >
<vec_elem val="685.0930273019305" pos="0" />
<vec_elem val="74.81968406307466" pos="1" />
<vec_elem val="1032.696020394059" pos="2" />
</vector>
</JointFrame>
<JointFrame...
</joint1-2>
</JointPositions>
</FinalJointPositions>

<MarkerPositions>     //Marker data: offsets, tracked positions, repositioned markers based on offsets and parameterized skeleton
<MarkerOffsets>     //Offsets of the markers relative to the center of mass of the markers placed on a limb(bone)
<Bone id="1" >
<vector length="3" id="0" name="Offset" >
<vec_elem val="-18.51422637891027" pos="0" />
<vec_elem val="80.22158642655715" pos="1" />
<vec_elem val="-61.92627443099342" pos="2" />
</vector>
<vector length="3" id="3" name="Offset" >
<vec_elem val="26.48784626154123" pos="0" />
<vec_elem val="18.01506651535026" pos="1" />
<vec_elem val="-5.237626582135748" pos="2" />
</vector>
<vector length="3" id="6" name="Offset" >
<vec_elem val="3.54961506921352" pos="0" />
<vec_elem val="-90.95700198353677" pos="1" />
<vec_elem val="58.94121610482424" pos="2" />
</vector>
</Bone>
<Bone ...
</MarkerOffsets>
<Frame FrameIndex="0" >     //for every frame the tracked position and the reconstructed one (corrected in version 1.1)
<Bone id="1" >
<Marker id="0" >
<vector length="3" name="FinalMarkerPosition" >     //reconstructed position (corrected in version 1.1)
<vec_elem val="424.7317975810138" pos="0" />
```

```
<vec_elem val="61.56459956171633" pos="1" />
<vec_elem val="1150.266772097681" pos="2" />
</vector>
<vector length="3" name="OrgMarkerPosition" >     //tracked position
<vec_elem val="601.187927246" pos="0" />
<vec_elem val="129.389923096" pos="1" />
<vec_elem val="1072.55993652" pos="2" />
</vector>
</Marker>
<Marker...
</Bone>
<Bone...
</Frame>
<Frame...
</MarkerPositions>
```

4.14 The Motion Capture Suit

During the development of the MoCap system two suits were used to capture motion data. The first version of the suit is a hand-crafted assembly of plastic plates with markers. The single pieces can be strapped on the actors' limbs by rubber bands (It can be seen in figure 4.3). The second suit is an off-the-self product of *3x3 Designs*.

The big advantage of the first suit is that most markers placed on a single limb stay in constant distance to each other. (Except for the ones placed on the breast, which has two plates and the hip, which on the other hand is rather solid itself) This makes clustering the markers easier. Furthermore the modular design of the suit allows it to adapt to different users. Also it is more comfortable to wear compared to a full suit, especially during the summer months. The markers, however, were placed relatively close together on the plates, which largely increased the probability of assigning a wrong id in the first step of the application flow. Increasing the size of the plates is difficult because then they would be too easily deformed or dislocated by skin and muscle movement. Since marker labeling still is a problem in the current version of the software now the other suit is used.

The second suit is a standard MoCap suit as used for movies and computer games. Markers can be flexibly attached to most body areas. As a basis for marker placement we used the positions described in [Dat07]. Additional markers were used to allow for some redundancy were it seemed necessary. The suit and final marker configuration can be seen in figure 4.4.

The standard deviation of distance between the markers is used for clustering them into limbs. Due to muscle and skin movement it can get as high as 1. (measurements taken in millimeter) This is acceptable and allows good segmentation

of the body. Therefore results shown in this work are mostly based on capture sessions using this suit.

Figure 4.3: Hannes Kaufmann wearing the first MoCap suit.

Figure 4.4: The author wearing the second MoCap suit.

Chapter 5

Results

This chapter will present the results of the MoCap program described in the earlier chapters of this thesis. Therefore the calculations of the main steps are analyzed and discussed in the next sections.

Note that the big advantage of the approach described in the chapters above is the flexibility to capture the motion of an arbitrary kinematic chain. Other, simpler, faster or more efficient methods exist for the human skeleton to be reconstructed. They, however, all lack flexibility, which is a prerequisite for many areas of application. In the next sections it can be seen how the ability of the system to adapt to non-human structures also helps evaluation. This primarily comes from the fact, that various types of input can be used for the program.

Three different classes of input were used to test the functionality of the system. On the one hand MoCap data has been simulated using 3D Studio Max. On the other hand an articulated structure consisting of three rods with markers was used to provide some real world measurement of joint distances. Finally the optical tracker described in section 2.5 was used to capture human data sets. For the latter the human body was either partially captured (e.g. upper body) or captured as a whole. The upper body data set has 21 markers placed on breast, back, hip and upper- as well as forearms and was captured for about 900 frames. The whole body data set consists of 28 markers placed on 10 body parts and is captured for almost 2000 frames. Whole body MoCap, however, has proven to be difficult for reasons described in the next section. Therefore results in this case only exist partially.

5.1 The Input data sets

First simulated data was used as an input to the application. It was modeled in 3D Studio Max and has the advantage of being more precise than captured data. Also ground truth measurements can be taken, which is rather difficult for a human skeleton. The animated skeleton consists of 13 bones roughly approximating the human body. On the skeleton are placed 31 virtual markers. The positions of

the markers are output for every frame of the animation and stored in a csm file. For test purposes two files with lengths 400 and 1500 frames were used. These file were then fed to the application.

A second class of input data sets was generated using an articulated rod construction consisting of three wooden rods connected by metal articulations. The length of each rod was about 35 centimeters. Three markers were placed on each of the parts using carbon fiber sticks attached to the rods. The marker positions were tracked over a sequence of between 400 and 500 frames.

Finally some data sets where captured using a MoCap suit with about 30 markers. Some data sets were obtained only with the upper part of the suit, while some were captured with the whole suit. For the whole body MoCap occlusions posed a great problem as will be explained in the following.

5.2 Results of Step 1: Temporal Marker Correspondence

Finding the temporal correspondence of the markers and labeling them accordingly has proven to be the most challenging task. Unfortunately at this point of time (December 2007) it still does not work perfectly. Due to the fact that only four cameras are used for the optical tracking many occlusions occur. These make it difficult to unambiguously identify a marker over the whole time of capture. This in turn is essential for the rest of the algorithms. Given the methods described in section 3.3 this is a very hard to accomplish goal. For some captured data-sets, however, the first step of the program produced sufficient results, which will be presented in the following. Also ideas for improvement of the marker labeling exist and will be described in short in chapter 6. A visualization of the result of step 1 (labeled markers) can be seen in figure 5.1.

The data sets obtained by simulation in 3D Studio Max contain markers, which are already labeled. Therefore they are not discussed in this section. For the articulated rod construction the marker labeling worked fine.

The first part of the temporal correspondence finding step as described in section 3.3 is labeling markers in successive frames. Two markers are assigned the same id if they don't move much from one frame to the other. This works rather well for both simulated and captured data as long as no marker gets occluded. However, this first step was replaced for this system by a prediction algorithm as described in [LaV03]. The method has been implemented in the tracker software. Thus it will not be described here in much detail. The basic idea is to predict the position of a marker for the next consecutive frames. This has two advantages over the method used before. Firstly, even in fast motions the id of the marker can be found in the next frame. Secondly, if the marker gets occluded for some frames - mostly markers only get occluded for a fracture of a second - the algorithm can find it and assign the correct id after it reappears. Our experiments showed that this works fine as long as markers are not placed

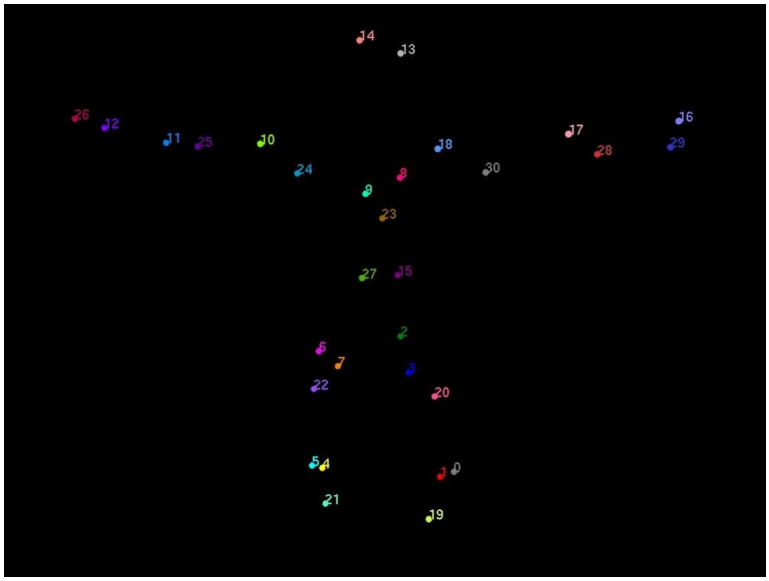

Figure 5.1: Markers labeled with ids placed on the model of a human body.

too close together on the suit. In this case marker ids of two close markers easily get switched, which is a hard to detect error. Finally it results in very bad joint calculations, although it might not even influence the process of assigning the markers to limbs as explained in the next section.

After the prediction algorithm has been run *Marker Sets* are created as described in sections 3.3 and 4.2. The second phase of the marker labeling task is to cluster the *Marker Sets* so each group corresponds to a physical marker. Here for this system lays the most likely point of failure. Since for a whole body MoCap data set there are created dozens or even hundreds of *Marker Sets* which have to be clustered into about 30 groups for the physical markers. Depending on the length of the captured sequence this can require a lot of computations since the distance has to be calculated between all the *Marker Sets*. Nevertheless for my test cases a capture sequence of about 400 to 2000 frames was usually sufficient to calculate the skeleton parameters. Thus the effort is somewhat limited. Also I can confirm the findings of Kirk et al. in [KOF05] that sampling produces almost identical results. Depending on the length of the *Marker Sets* and the number of poses therein even ten samples can be enough. In addition I am using a local approach, which further reduces calculation time and often makes the distance values more accurate. The basic idea of this algorithm is not to transform the whole body pose of the *Marker Sets* for comparison. Instead only a part of the body is transformed. Ideally for example an arm is rotated and translated into a local coordinate system based on the positions of a subset of markers placed on that arm. Since the arm is much more likely to pass through the same pose than

the whole body this should work much better. Unfortunately at this point of the application flow there is no information, which limb the markers are assigned to. However, taking markers that are close to the marker which the *Marker Set* belongs to also works fine in a lot of cases. For the test runs about 4-5 markers for a 30-35 marker suit produced the best result. Three markers produced too much errors due to skin movement and other influences, while more than 5 destroy the advantages of the local transformation.

As mentioned above the unambiguous identification of the markers is crucial for the correct calculation of the skeleton parameters. Therefore in many cases it is better to ignore a marker for a series of frames than to use it with a wrong id. This counts only if there are enough markers available and the capture sequence is long enough. I use some strategies to avoid wrong identification of markers by reducing the data. First all *Marker Sets* which are less than half a second of length are removed. There are two reasons for this. On the one hand it doesn't significantly improve the overall result. On the other hand a *Marker Set* this short is not very likely to contain more than a single pose. This means that the other *Marker Sets* that it is matched with have to have exactly that pose to produce a useful result. Otherwise the *Marker Set* gets clustered wrong and the whole system fails. For the same reasons *Marker Sets* which are longer (e.g. some seconds) but are not long enough to be useful without being clustered to other *Marker Sets* are removed in some cases. If the distance to every other *Marker Set* is larger than a threshold it gets deleted. (Currently 10 cm are used, which is about the minimum distance between markers on the MoCap suit)

Once the number of *Marker Sets* is reduced they are clustered into groups.(One group for each physical marker) I have tried multiple clustering algorithms for that purpose. First I tried Spectral Clustering as suggested [KOF05]. Since the parameter sigma is difficult to determine I also implemented and used a version with local sigmas as described in [ZMP04]. Also K-means (LLoyds, with a local search heuristic, a hybrid solution [KMN+02b] and K-means++ [AV07]) has been tested for this purpose. For the latter I implemented an exclusion list, which prevents overlapping *Marker Sets* to be clustered together. The best results were produced by Spectral Clustering with local sigmas. Also Spectral Clustering produced good results, when the correct parameters were used. The parameter sigma for that reason was inferred automatically by iterating it over a certain range. The clustering then was evaluated and the best clustering chosen. As an estimate for the quality of a clustering the distance between *Marker Sets* within the clusters as well as the distance of the *Marker Sets* of different clusters has been used. (The first has to be minimized while the latter should be maximized) Unfortunately in the current implementation the clustering sometimes produces errors for the captured data sets. It has proven especially tricky if multiple markers placed on the same limb get occluded at the (almost) same time. Due to the spatial proximity of the markers the distance values calculated between the two corresponding *Marker Sets* are also very close. This fact sometimes is the reason for markers to switch ids. This in turn, while not necessarily causing the articu-

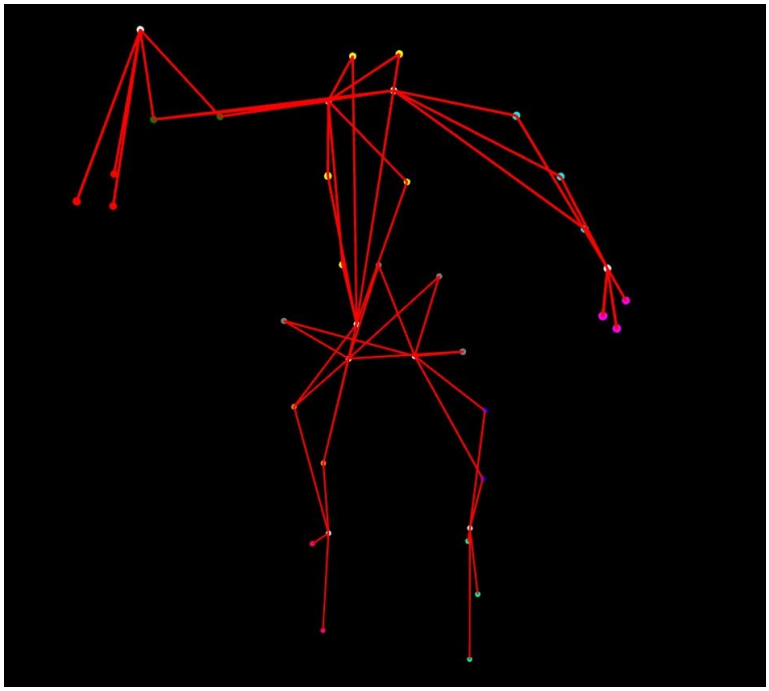

Figure 5.2: Skeleton produced from a dataset where markers switched ids. Especially the elbow on the left side of the image is reconstructed very badly.

lated structure to be wrong as a whole, results in a very crooked skeleton as can be seen in figure 5.2. Even worse than this phenomenon is the wrong assignment of ids to markers of different limbs. In that case most of the time no useful result can be produced by the subsequent steps.

5.3 Results of Step 2: Spectral Clustering

As mentioned before Spectral Clustering - or clustering in general - is being used at two points in the application flow. The first is the clustering of *Marker Sets,* which is needed to label the markers. Secondly, once the markers can be uniquely identified, they are clustered into groups consistent with the limb they are placed on. (Note that limb is used here more as a generic term, since for sake of flexibility the system doesn't (need to) know, whether the limb is an arm, leg, hip etc.) This section mainly focusses on the latter point of application for clustering, since the first was already discussed in the last section. A visualization of clustered markers can be seen in figure 5.3.

For the intended purpose it is rather difficult to classify the result of a clustering algorithm except into correct and incorrect. This is since a wrong grouping of limbs usually results in a false skeleton structure. (e.g. The upper arm con-

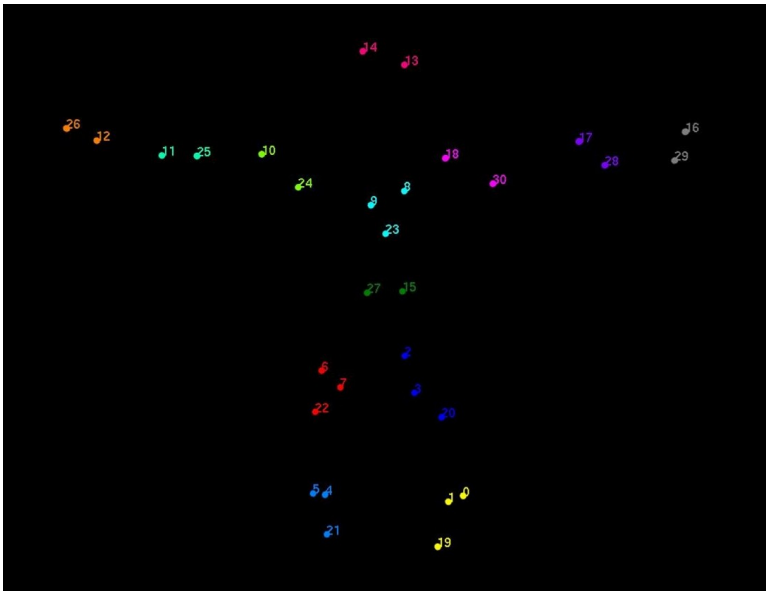

Figure 5.3: Visualization of clustered markers. (color indicates the cluster)

nected directly to the hip) For that reason the discussion of clustering algorithms will be rather short. The k-means algorithm used in the current version is k-means++ [AV07]. This is primarily for reasons of efficiency. Due to the good initialization the clustering with this algorithm usually needs less than a dozen iterations. Results with Lloyds and the hybrid solution described in [KMN+02b] are correct as well.

Simulated data has proven to be rather trivial to cluster. This comes from the fact that the distance between markers placed on a single limb does not change at all (except for numerical errors resulting from conversion into and from csm format etc.). Since the deviation from the average distance determines the cost values for the clustering the result is rather obvious. (For the cost function please refer to section 3.4) In our case the cost values between markers on the same limb and on different limbs differ by a factor of 10^6. For test purposes I implemented a greedy algorithm that simply looks for the lowest cost values in the cost matrix. Then it clusters the markers with the lowest cost together until the intended number of clusters is reached. For the artificial data sets this method works perfectly correct. For the captured data on the other hand Spectral Clustering again proofs to be the most reliable.

In the current version of the MoCap system Spectral Clustering with local sigma parameters is used. For calculation of the local sigmas five markers seem to work well. (See also section 4.9 parameter LOCALSIGMAK) It also is the most convenient solution since it doesn't require any user interaction. For captured data it usually produces correct results provided that the marker ids are correct.

Even in cases where marker ids on one or two limbs are switched the algorithm in many cases produces a correct marker-limb association. (Note that for a limb with only two markers it doesn't change the cost if the marker ids are switched) The cost values between markers on the same limb and on different limbs for captured data usually change by a factor of about 3 to 100. This of course depends on a number of factors. First of all the joint needs to be adequately exercised. Secondly, bad marker placement might result in wrong clustering. (e.g. Spectral Clustering had difficulties correctly assigning a marker placed on the shoulder - for most test runs it would cluster it with the upper arm) Also the number of samples used for the clustering strongly influences the result. While for the simulated data 3-5 samples produced correct results for the captured data sets about every fifth frame was used to calculate the cost matrix. Additionally skin- and muscle movement introduces certain errors. In general, however, clustering the markers into limbs is not a problem when using Spectral Clustering.

5.4 Results of Step 3: Joint Positions

Calculating the joint positions is the computationally most expensive part of the system. Especially minimizing the functions described in sections 3.5.1 and 3.5.2 requires most of the programs runtime, since there are thousands of unknowns involved. Therefore special effort has been taken to speed up these calculations. First of all it is crucial to initialize the joint positions to values that are close to the result to reduce the number of the iterations. Note that this should not only be done for efficiency. When using the origin as initialization for the joint positions the optimization algorithm got trapped in local minima for many joints. This is especially true for the simulated data, where the knee for example was modelled as perfect rotation axis. Thus the correct joint could lie anywhere on the axis as far as the minimization is concerned. In addition to the initialization the parameter α from the function in section 3.5.1 is being set to 0.1 to keep the joints close to the limbs it connects. This, however, introduces a small error because it is likely to move the joint away from its "perfect" position as I will explain later.

To speed up the joint calculation I used parallelization. The joint positions can be calculated independently since they don't influence each other by only reading the marker positions. Depending on the number of joints this can largely decrease computation time. I use parallelization at two points in the program. The first is for the joint calculations for the skeleton structure. Here hundreds of "joints" are calculated between all limbs. This is easy to distribute among the different processors. For the final joint positions it is not so efficient since only 10 to 15 joints have to be calculated. These calculations can not be well distributed since the computation time for the joints can differ significantly. However, the overall computation time is about cut in half by using parallelization on the test machine with four processor cores.

FTOL	10^{-6} is currently used for the break condition of the minimization algorithm.
ITMAX	100-250 iterations produced the best results. For simulated data usually less iterations were needed than for captured data.(Except for test purposes were nanometer precision was wanted)
ITMAXBRENT	Using 500 sometimes produces better results than the suggest default 100 from [PTVF92], but also slows down processing.
TOLBRENT	$2*10^{-8}$ produces good results
METHOD	Polak-Ribiere is choosen for the reasons outlined in section 3.6. With Fletcher-Reeves start positions seem to be not good enough because sometimes it doesn't converge.

Table 5.1: Parameters used for NLCG

The nonlinear conjugate gradient method(NLCG) was used to minimize the functions mentioned above. Since good minimization is crucial to the success of the system I will take a closer look at the parameters used. Table 5.1 shows the values that seem to work fine. For a more detailed description of the parameters please refer to section 4.9. Parameters not mentioned here have the default values specified in [PTVF92].

One way of evaluating the quality of a joint position is to look at the return value of the minimization function. Others will be discussed in the next section, where the skeleton's parameters are used to measure joint correctness. (Note that all measurements taken and discussed in the following are in millimeters.) A low return value of the minimization function tells us that the distances between the joint and the markers of the adjacent joints remains constant over the captured timespan. If this is the case we can assume that we found a good joint position. For the simulated data this return value is theoretically only limited by numerical precision, which is primarily limited by the output of the csm-export script. This only reads out floating point numbers with a mantissa size of six. (Also a break condition is used since the program should terminate in finite time) Due to the parameterization return values in this case can go as low as 10^{-7}. Sometimes, however, there is a local minimum the optimization algorithm gets trapped in, or roundoff errors strongly effect the return value. Especially for joints that are not exercised very much like the wrist or joints with less than six markers in the adjacent limbs this can happen. These local minima, however, usually are not a problem, because they are very close to the optimum (return values between 0.001 and 0.5) and accuracy in a submillimeter domain is not a goal for the system.

The captured data obviously can not perform as well, since numerous errors are introduced by the human body (skin and muscle movement, approximated

rotational joints) and to a lesser degree the tracker. Also approximating the spine by a single rotational joint introduces some error. However, return values are usually in a range of 2-20. This means that the average deviation of a marker to the joint is the square root of these numbers (in millimeter). For joints like the elbows, the shoulders, the hip and the knees the return values are generally more in the lower area (about 2-9) while for the back and the neck they are higher. (about 15) Due to the fact that the joints in the human body are only crude approximations of rotational joints this values can be considered sufficient. This is especially true for the shoulder and the knee, where the center of rotation moves depending on the current pose of the limb. (According to [SR05] the knee centers vary with up to 9 mm for adults and the shoulder, because of its complex nature, might deviate significantly stronger)

The return values produced by the optimization of our rod construction's joint positions are between 0.3 and 1.5. This also seems quite acceptable, considering the improvised joints and tracking errors.

The performance of the system is largely determined by the calculations of the joint finding optimization described here. The tests were run on a workstation with 2 AMD Opteron 280 dual core processors with 2.4 GHz. On this machine joint optimization took about five seconds for the rod construction. This is for 9 markers and three limbs captured for about 400 to 500 frames. The captured data set with 28 markers on ten body parts in about 2000 frames required up to one and a half minute. Note that the 2000 frames are after the cleanup, where frames are removed, which contain too little markers. For the artificial data with 1500 frames the optimization took about 70 seconds. Although for the simulated data sets more markers (31) and more limbs (13) were used than for the captured calculation time remains lower. This is not only due to the smaller number of frames. Also for some joints only very little iterations have to be calculated for acceptable results. These time values are taken using the parameters described in this section. Note that these can vary for different parameters. Especially the sampling, when determining the structure of the skeleton, has an important impact on the calculation time.

5.5 Results of Step 4: Skeleton Parameterization

Finally I will present the results of the last step, which is finding the skeleton's parameters. The quality of the output of this step very much depends on the correct findings of the joint optimization in the last step. Bad optimization values there usually result in a skeleton that has limb lengths that strongly deviate from the ones of the captured subject. Unfortunately measuring limb lengths on a human skeleton is not an easy thing to do. Therefore lengths measurements are only taken from simulated data and from the rod construction. As a universal measurement of correctness, which can be applied to all the data sets, the parameterized skeleton is fitted back to the captured data. This is done using

the inverse kinematics algorithm described in 3.8.2. Then the average distance between the reconstructed marker positions and the original ones is measured and averaged.

For a start evaluation of the simulated data is presented. As said before precision of the output here is largely limited by the numerical precision of the input. Also rotation axis pose a certain problem as can be seen later. The evaluation of two csm files is shown here to prove the correct functionality of the skeleton parameterization algorithms. The data sets contain the marker positions of the skeleton described at the beginning of this chapter over a period of 400 and 1500 frames. For the algorithm probably less frames (or samples of the animation sequence) also would be sufficient. However, I wanted to keep the input comparable to the captured data set and therefore tried to animate the skeleton with reasonably lifelike motions at a realistic speed. Tables 5.2 and 5.3 show the average deviation of the marker positions from the positions in the csm file after the skeleton has been fitted back to the data. From that values multiple conclusions can be drawn. First of all the values given there strongly indicate that the parameterized skeleton almost perfectly fits the one animated in 3D Studio Max. Since the deviations are very low for most limbs joint offsets and marker offsets have to be correct in order to allow the IK algorithm to fit them back to the data. Secondly as described in section 3.8.2 the skeleton is adjusted to the data hierarchically. This means that errors of the IK algorithm propagate along the limbs as they are fitted. (e.g. starting at the breast followed by upper arm, forearm and hand) The increase of deviation from the torso to the outer extremities therefore not only stems from numerical inaccuracies or calculation errors but also from the IK algorithm. The third conclusion that can be drawn from the data is the fact that marker placement strongly affects the result of the parameterization. Thus the "bad" reconstruction value for the right hand that can be seen for all three animations is from a slightly unfortunate marker placement. As can also be seen in figure 5.1 marker 16 and 29 are placed very close together producing the criticized result. Note that for the simulated data markers always maintain their ids because it is exported with the data from 3D Studio Max.

To show that the above method provides a good means to measure the quality of skeleton parameterization the joint positions of the 400 frame animation were also evaluated. Therefore the joint positions were exported from 3D Studio Max and compared to the findings of the MoCap system. This was done after calculating the joint positions in step 3 and after the skeleton was parameterized and fitted back to the data. The result can be seen in table 5.4. Comparing the two righter columns of this table clearly shows an improvement, after the parameterization for almost every joint. The largest deviation is produced by the knees, which can be considered a special case. This is because they are modelled by a rotational axis. Thus any point on the axis can be a perfect joint as far as the optimization algorithm is concerned. (Although a position close to the markers is favored) For that reason a deviation of 10 can be considered a good value. This

Limb	Marker	Average deviation
Breast	8	0.0009944471620018838
	9	0.0009854093906319728
	23	0.0009929703796803173
Head	13	0.001053717895964039
	14	0.0010885129095937198
R. Upper Arm	18	0.019372854501369687
	30	0.019877490959252953
R. Forearm	17	0.02619963194425512
	28	0.03013364570837896
R. Hand	16	1.1682378545645293
	29	1.6265735381409094
L. Upper Arm	10	0.001021008935532011
	24	0.0010259126044370166
L. Forearm	11	0.16695957608696657
	25	0.3544017028244262
L. Hand	12	0.38819061525847887
	26	0.41291686881058676
Hip	15	0.001039108117499345
	27	0.0010261218304355028
R. Thigh	2	0.05226658780682448
	3	0.05105508606809256
	20	0.04878096073048679
R. Lower Leg	0	0.055946226356728634
	1	0.054645184870330786
	19	0.052358214328769234
L. Thigh	6	0.2759348371328221
	7	0.2536390742656726
	22	0.2615930451353708
L. Lower Leg	4	0.18809646096329097
	5	0.20223278213782447
	21	0.19222953292862388

Table 5.2: Average deviation between the original and reconstructed marker positions for the simulated dataset with 1500 frames

counts especially since visual inspection shows that the joint does stays in place along the rotational axis and is therefore not troublesome for e.g. animation purposes. Comparing the values of tables 5.3 and 5.4 also produces some interesting findings. Leaving the knees aside there can be observed a correlation between the values of the two tables. So the right hand/wrist has the worst deviation followed by the left hand/wrist and the right forearm/elbow. The breast/spine head/neck thigh/hip joint combination on the other hand show only minimal error.

Limb	Marker	Average deviation
Breast	8	0.0009993540877580689
	9	0.00099137673909368
	23	0.0009943304356966772
L. Upper Arm	10	0.0010337998665188885
	24	0.0010379928463720046
L. Forearm	11	0.004284588521316699
	25	0.007746556963900375
L. Hand	12	0.25100862029414855
	26	0.18982405094275848
Head	13	0.001055851087810452
	14	0.0010663656720730033
R. Upper Arm	18	0.12197313118800823
	30	0.16017175191713812
R. Forearm	17	0.21883503340144853
	28	0.22195148959048136
R. Hand	16	1.7186381536680644
	29	1.5898171821315759
Hip	15	0.0010695627215237399
	27	0.0010346725509701694
R. Thigh	2	0.0010538344234221817
	3	0.0010571968961394665
	20	0.001045088599133401
R. Lower Leg	0	0.001056190230784203
	1	0.0010708357153337769
	19	0.0010451868546563603
L. Thigh	6	0.001060967788309057
	7	0.0010710289045731012
	22	0.0010780554763261533
L. Lower Leg	4	0.0010653798965889025
	5	0.001084343795232593
	21	0.0010621389667053581

Table 5.3: Average deviation between the original and reconstructed marker positions for the simulated dataset with 400 frames

In this paragraph a closer look will be taken a the results of the rod construction depicted in tables 5.5, 5.6, 5.7 and 5.8. Note that these data sets were captured using the tracker described in [Meh06], which was later upgraded to what is described in [PK07]. Unfortunately there wasn't time for a second capture session with the new tracker, which would have allowed for higher frame rates and more accuracy. The results, however, are nevertheless acceptable with average deviations of 0.2 to 3 millimeters. Again it can be observed that the ac-

Joint	Average Deviation	Average Deviation after Param.
Spine	0.000569756	0.000575338
Right Elbow	1.82806	0.338819
Right Hip Joint	0.000834624	0.000633745
Right Knee	9.28491	9.28484
Right Shoulder	1.29147	0.520079
Right Wrist	12.0792	4.44404
Neck	0.000704843	0.000814122
Left Elbow	0.0976497	0.0216419
Left Hip Joint	0.00202592	0.000974378
Left Knee	9.30966	9.30965
Left Shoulder	0.000571244	0.000549666
Left Wrist	5.26235	1.40935

Table 5.4: Average Deviation of the calculated joint positions from the original joint positions. The second column shows the deviation after step 3, while the third column presents the deviation after step 4. The values are taken from the 400 frame simulated dataset.

Limb	Marker	Average deviation
2	0	0.4147361012924929
2	5	0.37781916256059556
2	7	0.26672217942744125
0	1	1.5963650926389057
0	2	2.0352633099941686
0	3	1.6813691979813832
1	4	1.9305034533459826
1	6	2.3899574574708047
1	8	2.4588358414915517

Table 5.5: Average deviation between the original and reconstructed marker positions for the captured rod construction. (First recording)

curacy for the centerpiece (the root) is better than the ones for the outer pieces. In Addition to the deviation of the markers the length of the middle rod was measured and compared as by Kirk et al. in [KOF05]. The measured length is 359 millimeters +-1.5 millimeters for the slackness of the articulations. The values calculated by the algorithm can be seen in table 5.9. The average distance between the two joints was computed to be 360.46 millimeters, which is rather close to the original length, when considered the slackness. (For a more precise statistical evaluation more data sets would have been necessary)

Finally the evaluation of the captured human motions is being presented. Two data sets were processed to test the functionality of the system. Due to the marker labeling problems pointed out earlier no complete correct data set of

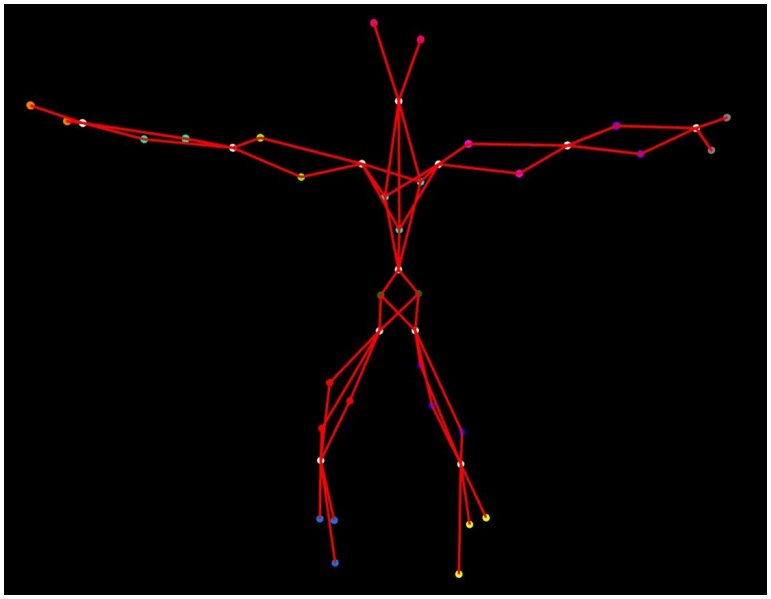

Figure 5.4: Visualization of a parameterized skeleton. The joints(white spheres) are connected to the markers(colored spheres) of the adjacent limbs.

Limb	Marker	Average deviation
0	4	0.4930096306771767
0	6	0.8462943467045175
0	7	0.6356741256914126
2	2	1.8171900838161508
2	3	1.983348238076757
2	8	2.045353616876127
1	0	2.21632336100837
1	1	2.1025654559648967
1	5	2.3977470500103215

Table 5.6: Average deviation between the original and reconstructed marker positions for the captured rod construction. (Second recording)

whole body capture was produced. I will nevertheless use one of the data sets produced here to point out how the system gets along with erroneous marker ids. The second data set described in the following paragraphs consists of a captured torso, which has correct marker labeling.

The whole body data set contains three errors related to marker labeling. One is in the breast the other is in the right thigh, while the third is in the left forearm. In all cases markers switch ids for a certain amount of time. This, however, in this case does not result in wrong clustering of the markers in step 2. For step 3 and 4, the joint calculation and the parameterization the effect is quiet

Limb	Marker	Average deviation
1	1	0.8709467629047596
1	3	0.5883733946165258
1	4	0.855782477316303
2	2	2.420136676821549
2	6	3.2790877180064992
2	8	2.5413330360419484
0	0	2.3121023866948653
0	5	1.9858107545649495
0	7	2.4726080992295763

Table 5.7: Average deviation between the original and reconstructed marker positions for the captured rod construction. (Third recording)

Limb	Marker	Average deviation
1	1	0.4835725362370122
1	2	0.5783252175350256
1	7	0.4668084807516744
2	4	1.880682268129612
2	5	2.1169604998853453
2	8	2.370866398233638
0	0	2.6072404299729732
0	3	1.9735770404335953
0	6	2.259741558746566

Table 5.8: Average deviation between the original and reconstructed marker positions for the captured rod construction. (Fourth recording)

Measurement	Value (mm)
Measured Distance	359 +-1.5
Data set 1	365.03
Data set 2	359.03
Data set 3	360.97
Data set 4	356.80
Average of Data set	360.46

Table 5.9: Measured and calculated distances between the two joints of the rod linkage

dramatically. The biggest problem is, that one switch occurs between markers placed on the breast. Because of that joint values between the breast and the other limbs are rather bad. Therefore the spanning tree algorithm doesn't find the breast limb to be the root. Instead the left upper arm becomes root and is connected to the hip. The other errors only have local effect on the left elbow and

Limb	Marker	Average deviation
L. Upper Arm	5	2.4022750586371466
	18	2.402280803122446
R. Upper Arm	6	131.48548639516954
	8	125.83802232848232
	25	125.07409100557012
R. Forearm	4	116.38256615519327
	13	123.22801348474934
	24	119.44519064558801
Breast	7	94.72896264401112
	15	78.39185808420885
	22	96.87829236078828
	23	102.55187195139979
	26	99.03937150775894
L. Thigh	14	52.52161211225093
	16	56.03526577878675
L. Lower Leg	3	48.188698176736644
	12	85.64028591349198
Hip	9	44.3944565368421
	11	136.52220363882273
	17	72.65360401700173
R. Thigh	10	103.99633485200287
	19	86.88035651295951
R. Lower Leg	1	89.02004073845008
	2	94.25114722777029
	27	79.58357849740945
L. Forearm	0	53.44981708249444
	20	69.59897715470238

Table 5.10: Average deviation between the original and reconstructed marker positions for the captured whole body dataset

the right knee. This only leaves the left leg and right arm untouched. However, due to the error propagation in the IK algorithm the deviations here too are in the centimeter domain as can be seen in table 5.10.

The last data set for which the results are evaluated is that of a captured upper body (with arms). Figure 5.5 shows the parameterized skeleton together with the reconstructed marker positions. From visual inspection it appears to be correct, except the joint approximating the spine shows to be too much on the right side. This, however is a result of the fact, that the spine is hardly exercised during the captured time span. Otherwise again it can be observed that the errors sum up at the outer extremities, while the root has relatively good distance values. With distance values between 4 and 22 millimeters ultimately the parameterization can

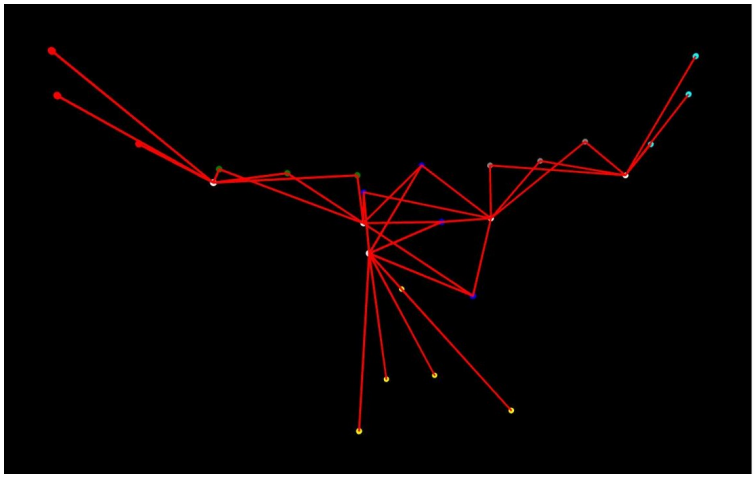

Figure 5.5: Visualization of the skeleton produced for a captured upper body.

be considered successful.

Limb	Marker	Average deviation
Breast	8	4.6598545027611324
	11	4.896348347319935
	13	5.479694854741079
	16	4.112358674976455
Hip	1	4.800415751645614
	2	6.892094587665252
	3	6.101659467333794
	12	5.287672032688435
	14	5.490909491832862
L. Upper Arm	5	5.762807038418709
	6	6.0897567003876025
	19	8.203250144989813
L. Forearm	7	14.651537550944884
	10	12.942072550283347
	20	14.235432796730349
R. Upper Arm	0	6.956793875333789
	4	8.749547061568276
	9	13.422635816712248
R. Forearm	15	22.93096995363695
	17	22.239811825942017
	18	20.3775491160917

Table 5.11: Average deviation between the original and reconstructed marker positions for the captured upper body dataset

Chapter 6

Future Work & Conclusion

6.1 Improvement of Marker Labeling

Since assigning ids to the markers (i.e. clustering the *Marker Sets*) still is a problem in the current version of the system I'm trying to improve it. One method I am currently implementing is to merge the first two steps of the system into one. Then using information that is only available after clustering the markers into limbs for the marker ids.

The basic idea is to iteratively improve the id labeling and the limb association using the following algorithm:

1. Cluster the *Marker Sets* (MS) into *Merged Marker Sets* (MMS) using the minimum distance D_{ij} described in section 3.3. This is similar to the original step1 except there can be more MMSs than physical markers.

2. Cluster the MMSs into *Limb Merged Marker Sets* (LMMS) using the variance of distance $varDist_{ij}$ [1]from section 3.4. This is analogue to step 2 in the original algorithm.

3. Iterate over the LMMSs and find the LMMS l_x for which the $varDist_{ij}$ [2]is largest between two MMSs mms_i and mms_j. Then find the MS(s) responsible in mms_i and mms_j and remove them to a pool of MMSs P.

4. Try finding MMS(s) were one/(some) member(s) of P can be fit into so that $varDist_{ij}$ does not get significantly increased for all the pairs mms_i and mms_j in the corresponding LMMS.

5. If the overall $varDist$ is below a certain threshold stop. Otherwise goto 3.

[1]Note that $varDist_{ij}$ is calculated from markers in the same frame, while D_{ij} is computed from Marker Sets that don't overlap in time

[2]Instead of $varDist_{ij}$ I am using the difference of the distance of the MSs in mms_i and mms_j because $varDist_{ij}$ is computationally expensive. That in theory should produce a similar result.

Unfortunately I couldn't finish the implementation of this method yet and thus can not say for sure how much it improves the labeling. The effect of the algorithm also is depending on the number of markers on the limbs. The two main advantages it should have are:

- Detect if two MS belonging to different limbs are clustered together and correct it.
- Given there are two or more markers on a limb and at least one of the markers has an id. Then the $Dist_{ij}$ to that marker can help to identify other MSs on that limb. Or in other cases it might rule out that a MS belongs to a certain limb.

The method described above does not help to avoid markers switching ids when on the same limb if there is only two of them, because the $Dist_{ij}$ stays the same if they do. Also it might generate some computational overhead. However, I'm quite confident it will make the system more flexible and reliable.

6.2 PlayMancer & Real Time

PlayMancer is an EU project which has the goal of developing a *Serious Game* environment. A main goal of the project is to support the development of *Universally Accessible Games*, which for example can be used for physical rehabilitation. Among other modes of user interaction it will contain a module for player motion tracking. This will be developed by the Virtual Reality Group of the Institute for Software Technology and Interactive Systems at Vienna University of Technology. As a part of this group I will adapt and improve the system described in this thesis to generate and parameterize the skeleton of players. For PlayMancer usability, reliability and performance are crucial design goals. Therefore it is not yet clear if the improvements described in section 6.1 are sufficient or if the flexible approach described in this thesis has to be given up. Using a predefined skeleton, which is adapted to the users proportions, would be one way to accomplish the above goals. Also real-time parameterization could be interesting in this context. The algorithms described in this thesis do not work in real time since they require a certain amount of captured movement data. Also some time is needed to process the data. This is a drawback for areas of application, were results are needed fast. (like for example PlayMancer) An approach like in [CL05] might help to achieve results faster. Which of the above methods finally will be used, however, is yet uncertain.

6.3 Conclusion

In this work the theoretical foundations necessary for the generation of a skeletal structure from optical motion capture data have been discussed. This includes a

small survey of tracking methods, their properties and the mathematical foundations of the algorithms used.

Based on these a software application to retrieve the skeleton parameters was designed. Design considerations were made for the program to be used in a teaching and research environment. Comparison to algorithms used in other MoCap systems have been drawn where it seemed interesting. The design also specifies the application flow as data is passed from stage to stage.

Also the implementation of the MoCap software has been documented during this work. The main part of the implementation is the four stages from marker labeling, marker clustering, joint calculation and estimation of the skeleton structure. This includes the algorithms used in these steps. Furthermore the properties of the graphical user interface and the input and output formats have been described. Additionally an inverse kinematics algorithm was implemented, which allows to fit the parameterized skeleton back to the captured data.

The software implemented was also thoroughly tested using simulated and captured data. For that purpose a low cost optical tracking system with passive markers was used. The results, which have been presented, show that the software works if during the tracking process markers don't get occluded too often. Then the algorithms find the correspondences between tracked and physical markers over the whole captured time span. This is a prerequisite for correct computations in the other steps.

If the markers can be identified they are correctly grouped into limbs as has been documented. Also the joint optimization algorithm has been proven to work fine. Finally the parameters calculated have been shown to be within an acceptable range of accuracy. Therefore different data sets and methods of measurement were used. Due to the lack of ground truth measurements for the captured data a new measurement method was introduced using the deviation of tracked and reconstructed marker positions. Results of the evaluations have been presented for each data set. Additionally the performance of the software has been analyzed including the improvements of multi processor usage.

Before it can be expected to produce stable results for whole body MoCap, however, some improvement to the marker labeling has to be implemented. Due to a lack of time, this has not been done during this work. Nevertheless suggestions for enhancements have been made in the above sections and the author is optimistic that these issues can and will be resolved in future work.

Literaturverzeichnis

[AV07] David Arthur and Sergei Vassilvitskii. k-means++: the advantages of careful seeding. In *SODA '07: Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035, Philadelphia, PA, USA, 2007. Society for Industrial and Applied Mathematics.

[BC03] Grigore C. Burdea and Philippe Coiffet. *Virtual Reality Technology.* Wiley-IEEE Press, 2003.

[BKL04] Doug A. Bowman, Ernst Kruijff, and Joseph J.and Ivan Poupyrev LaViola, Jr. *3D User Interfaces: Theory and Practice.* Addison-Wesley, 2004.

[CL05] Jonathan Cameron and Joan Lasenby. A real-time sequential algorithm for human joint localization, 2005. http://www-sigproc.eng.cam.ac.uk.

[Dat07] CMU Graphics Lab Motion Capture Database. Marker placement guide, 2007. pdf date. http://mocap.cs.cmu.edu/markerPlacementGuide.pdf.

[Dav06] R. B. Davies. Documentation for newmat10d, a matrix library in c++, April 2006. http://www.robertnz.net/nm10.htm.

[DGK04] Inderjit S. Dhillon, Yuqiang Guan, and Brian Kulis. Kernel k-means, spectral clustering and normalized cuts. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.

[Fur99] Maureen Furniss. Motion capture. MIT communications forum, December 1999. http://web.mit.edu/comm-forum/papers/furniss.html.

[GF05] Gutemberg B. Guerra-Filho1. Optical motion capture: Theory and implementation. *Journal of Theoretical and Applied Informatics (RITA)*, 12(2), 2005.

[Gor06] Devin Gordon. Polar expedition. *Newsweek*, Oct. 25, 2006. http://www.msnbc.msn.com/id/6262593/site/newsweek/.

[Hor87] Berthold K. P. Horn. Closed-form solution of absolute orientation using unit quaternions. *Journal of the Optical Society of America*, 4, 1987.

[JFGT02] Moon Ryul Jung, Ronald Fisher, Michael Gleicher, and Jeffrey A. Thingvold. Motion capture and motion editing: From observation to animation. 2002. http://www.cs.wisc.edu/graphics/Papers/Gleicher/MocapMusings/.

[JMF99] A. K. Jain, M.N. Murty, and P. J. Flynn. Data clustering: A review. *ACM Computing Surveys*, 31(3), September 1999.

[Kin05] V. Kindratenko. Calibration of electromagnetic tracking devices. *Virtual Reality: Research, Development, and Applications*, 4(2):139–150, June 2005.

[KMN+02a] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. Efficient algorithms for k-means clustering, 2002. http://www.cs.umd.edu/ mount/Projects/KMeans/.

[KMN+02b] Tapas Kanungo, David M. Mount, Nathan S. Netanyahu, Christine D. Piatko, Ruth Silverman, and Angela Y. Wu. An efficient k-means clustering algorithm: Analysis and implementation. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 24(7), 2002.

[KOF05] Adam G. Kirk, James F. OBrien, and David A. Forsyth. Skeletal parameter estimation from optical motion capture data. In *IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2005 Volume 2*. IEEE Conf. on Computer Vision and Pattern Recognition (CVPR), 2005.

[LaV03] Joseph J. LaViola. Double exponential smoothing: an alternative to kalman filter-based predictive tracking. In *EGVE '03: Proceedings of the workshop on Virtual environments 2003*, pages 199–206, New York, NY, USA, 2003. ACM.

[Meh06] Michael Mehling. Implementation of a low cost marker based infrared optical tracking system. Master's thesis, Fachhochschule Stuttgart- Hochschule der Medien, 2006.

[Men99] Alberto Menache. *Understanding Motion Capture for Computer Animation and Video Games*. Morgan Kaufmann Publishers Inc, 1999.

[Mey00] Carl D. Meyer. *Matrix Analysis and Applied Linear Algebra*. Society for Industrial and Applied Mathematics, 2000.

[Mot07a] Meta Motion. Clinical motion analysis, 2007. http://www.metamotion.com.

[Mot07b] Meta Motion. Who uses motion capture, 2007. http://www.metamotion.com/motion-capture/motion-capture-who-1.htm.

[MS00] Marina Meila and Jianbo Shi. Learning segmentation by random walks. In *Neural Information Processing Systems*, pages 873–879, 2000.

[Mül03] Olaf Müller, 2003. http://www.informatik.uni-osnabrueck.de/olaf/.

[NJW02] Andrew Y. Ng, Michael I. Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. *Advances in Neural Information Processing Systems*, 14, 2002.

[NMFP98] M. Nixon, B. McCallum, W. Fright, and N. Price. The effects of metals and interfering fields on electromagnetic trackers. *Teleoperators and Virtual Environments*, 7(2):204–218, 1998.

[Pha07] PhaseSpace, November 2007. http://www.phasespace.com.

[Pin07] Thomas Pintaric. iotracker, 2007. www.iotracker.com.

[PK07] Thomas Pintaric and Hannes Kaufmann. Affordable infrared-optical pose-tracking for virtual and augmented reality. In *Proceedings of Trends and Issues in Tracking for Virtual Environments Workshop*. IEEE VR 2007, 2007.

[PTVF92] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C*. Press Syndicate of the University of Cambridge, 1992.

[RDB01] Jannick P. Rolland, Larry D. Davis, and Yohan Baillot. A survey of tracking technology for virtual environments. Technical report, School of Electrical Engineering and Computer Science, University of Central Florida, 2001.

[RL02] Maurice Ringer and Joan Lasenby. A procedure for automatically estimating model parameters in optical motion capture. *Image and Vision Computing*, 22(10), 2002.

[Sed84] Robert Sedgewick. *Algorithms*. Addison-Wesley Publishing Co, 1984.

[She94] Jonathan Richard Shewchuk. An introduction to the conjugate gradient method without the agonizing pain, August 1994. http://www.cs.cmu.edu/ quake-papers/painless-conjugate-gradient.pdf.

[SPB+98] M.-C. Silaghi, R. Plankers, R. Boulic, P. Fua, and D. Thalmann. Local and global skeleton fitting techniques for optical motion capture. *IFIP CapTech 98, Geneva*, 1537:26–40, 1998.

[SR05] Michael H. Schwartz and Adam Rozumalski. A new method for estimating joint parameters from motion data. *Journal of Biomechanics*, 38:107–116, 2005.

[Sta02] Kay M. Staney. *Handbook of Virtual Environment Technology,*. Lawrence Erlbaum Associates„ 2002.

[TGB00] Deepak Tolani, Ambarish Goswami, and Norman I. Badler. Real-time inverse kinematics techniques for anthropomorphic limbs. *Graphical models*, 62(5):353–388, 2000. citeseer.ist.psu.edu/tolani00realtime.html.

[Tro07] Trolltech, 2007. http://trolltech.com/products/qt/.

[Val06] Anna Vallgårda. Principles of positioning, February 2006. http://www.itu.dk/people/schmidt/teaching/lma/.

[Vic07a] Vicon. Mainframe entertainment leverages vicon to bring skateboard tale to life for tony hawk in boom boom sabotage, 2007. http://www.vicon.com/company/releases/062106.htm.

[Vic07b] Vicon. Vicon, November 2007. http://www.vicon.com.

[Vil04] J. Villanueva. The cubic and quartic equations in intermediate algebra courses, 2004. http://archives.math.utk.edu/ICTCM/EP-16/C37/pdf/paper.pdf.

[vL06] Ulrike von Luxburg. A tutorial on spectral clustering. Technical Report 149, Max Planck Institute for Biological Cybernetics, August 2006.

[Wei04] Christian Weisel. Entwicklung eines markerbasierten motion capturing-systems für echtzeitanwendungen. Master's thesis, Fachochschule Gießen Friedberg, 2004.

[WF02] Greg Welch and Eric Foxlin. Motion tracking: No silver bullet, but a respectible arsenal. *IEEE Computer Graphics and Applications*, 22, 2002.

[YS03] Stella X. Yu and Jianbo Shi. Multiclass spectral clustering. In *Proceedings of the Ninth IEEE International Conference on Computer Vision*, 2003.

[ZMP04] Lihi Zelnik-Manor and Pietro Perona. Self-tuning spectral clustering. *Neural Information Processing Systems*, 2004.