2009:142 CIV

MASTER'S THESIS

Inverse Kinematics

Rickard Nilsson

Luleå University of Technology

MSc Programmes in Engineering
Computer Science and Engineering
Department of Computer Science and Electrical Engineering
Division of Computer Science

2009:142 CIV - ISSN: 1402-1617 - ISRN: LTU-EX--09/142--SE

Preface

The work for this thesis was conducted under the fall of 2007 at Agency9 AB in Luleå for Luleå University of technology - Computer Sciene and Engineering. Agency9 wanted an evaluation for some of the most used methods that can solve inverse kinematics.

I would like to thank all the guys at the office, Tomas, Khashayar, Thomas, Lasse, Johan, Martin and Richard at Agency9 in Luleå, who have listened and answered many questions.

Abstract

The computer is getting more powerful every day and creating animations for a 3D model "on the fly" rather than pre-made animations is getting more feasible. A 3D model that is composed by segments and joints, formed for example like a human, can also be viewed as a kinematic model, the joints can be modified such that different postures can be achieved for the model. A kinematic model have a hierarchic tree structure with a parent-child relationship. A change made to a parent propagates to its children. Interacting with a kinematic model can be done in two distinct ways, forward kinematics and inverse kinematics.

The most common way of interacting with a kinematic model is by using forward kinematics, which works by changing each joint until a desirable posture is achieved. The other way, known as inverse kinematics is more complex and needs more computer resources, only need a direction or a position for any joint of where it should be.

The inverse kinematic methods are all using a matrix called Jacobian matrix, which consists of all first-order partial derivates derived from a kinematic model.

The main goal of this thesis is to evaluate current methods solving Inverse Kinematics and to see if it possible to create animations "on the fly" for a model.

Contents

Chapter 1

Introduction

The performance of a computer is increasing every year and game producers have the ability to use it more extensively, not only for plain graphic improvements but also for other algorithms. Algorithms for creating animations "on the fly" that look natural for an animal or human with the ability to freely interact with the surrounding environment is hard to do with current animation techniques. Currently, most animations are pre-generated with few possibilities of beeing able to alter it. Special problems arises when there are dynamical changes in the enivronment. For instance if there is a pre made animation of a human arm reaching a door knob, the animation has to be made in consideration of exactly where the objects are positioned. If the door shrunk to half its size or just were moved further away the given animation would not change it self accordingly.

To compensate for the dynamical changes in the environment a diffrent approach is needed. Given a human arm model, which consists of a segments and joints connected to each other. By calculating the angular scalars for each joint such that the palm of the hand is as close it can get to the door knob would compensate for moving the door knob. This method is called Inverse Kinematics (IK) and it is a way of making articulated models, for example a human or an animal model, to move more freely within its boundaries and be able to interact in a very different manner.

1.1 Agency9 and AgentFX

Agency9 AB[1] was founded in 2003 with the vision to "make high-performance three dimensional visualization, 3D, available to everone and everywhere". At that time an advanced platform for real-time rendering of 3D graphics called AgentFX had been developed. AgentFX is basically a platform-independet 3D engine written in Java using OpenGL, which also supports loading the open

[1] www.agency9.se

schema for digital-asset, known as COLLADA (see section 3.1), for creating scene-graphs. The AgentFX engine is tightly connected to the COLLADA-structure [14].

1.2 Problem formulation

There are two distinct methods of solving inverse kinematics, analytical and iterative. The iterative method gives the solution by solving an approximation of the system, and by updating the system with the output from the solver each iteration until it converges. The analytical method solves the whole system at once.

The major part of this thesis will examine and evaluate several methods of solving inverse kinematics for articulated models and the possibility to use it together with AgentFX. It will mainly be focused on iterative methods, but also give a short survey of the analytical approach.

An other large part of this thesis will be to create a scene-graph-like structure for the IK and to understand the properties of a scene-graph, such as how transformations of rotation and translation works with inverse kinematics.

The final implemented solution must be general and dynamic in regard to the variety of models it shall be able to handle. A major problem to consider is that the system has to be robust since it's known that some methods that solves inverse kinematics becomes instable in some situations. The instability will be reflected in the articulated model as fast and undesirable movements.

The system also has to be fast enough such that slower computers will be able to handle the algorithm, which can be very complex.

Another important thing to point out is that the inverse kinematic application shall not be integrated within the AgentFX 3D-engine, it should only have the ability to control it.

1.3 Delimitations

Since the IK solution shall not be integrated within the AgentFX 3D-engine it has to have its own tree structure much like a scene graph with node properties as translation and rotation transformations. Content created by the IK application can never be stored in a COLLADA file since it is not defined in its standard.

As the algorithms of inverse kinematics are rather complex in nature, only a few algorithms using the Jacobian matrix will be implemented and tested.

Chapter 2

Theory

This chapter introduces the fundamental principles of kinematics and how forward and inverse kinematics are defined. Further will it present different methods for solving inverse kinematics.

2.1 Kinematics

Kinematics is defined as "study of motion: a branch of physics that deals with the motion of system without reference to force and mass" [1]. A kinematic model is built with segments and joints in a hierarchic structure with a parent-child relationship.

Segments are considered as rigid bodies and joints are the link between the segments. By rotating a joint around an arbitrary axis that coinside within the joint, all its children will also rotate around the same axis since their relationship is set in that way that it inherits all its parents transformations. This can be seen in Figure 2.1 and Figure 2.2.

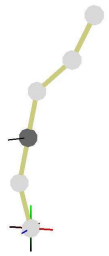


Figure 2.1: *A kinematic model, where the small spheres are joints.*

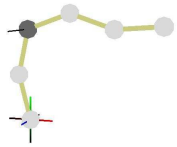


Figure 2.2: *Rotation around the axis marked black that coincide within the grey joint.*

Joints can be given the ability to rotate and translate, if a joint can rotate around one axis it is said to have one degree of freedom. The same is for translations.

2.1.1 Degree of Freedom

The number of degrees of freedom a joint can have is directly related to how many dimensions the joint is in. In three dimensions the joint can have at most three rotational and three transitional degrees of freedom, where the rotational and transitional axises are orthogonal sets.

In short the degree of freedom explains how a joint can move in their space. Typcial joints in three dimensions are: hinge, pivot, universal and ball joints.

Hinge and pivot joints have one degree of freedom, pivot joints are restricted to rotate along an axis towards its child. Universal joints are based on two hinge joints with 90 degree displacement of eachother, which means that it can bend in any direction. The ball joint have three degrees of freedom and works like the universal joint with the addition that it can rotate along an axis towards its child.

Transitional joints are easier to define since they can only move along a given axis. That is with one degree of freedom it can move along a line, with two it can move in a plane and with three it will be able to move in a room.

For simplicity regarding notation this thesis will only handle rotational joint but the theory works for transitional joints aswell. The orientation of a rotational joint is decided by real scalars that is in the case of rotational joints the angle of a joint.

2.1.2 Constraints

Consider a kinematic model looking like a human, constraining certain joints are needed in order to avoid undesirable postures. For example a knee, where the joint is a type of hinge joint with one degree of freedom. However a constraint must be added since a knee can not rotate freely around its rotation axis. The constraint must then be set to have a maximum and a minimum value of the scalar in the joint where it is allowed to rotate within as in Figure 2.3. For rotational joints the scalar is the angle.

2.2 Forward Kinematics

There are two different methods of controlling kinematic models that are used in this thesis known as forward and inverse kinematics [4].

The forward kinematic problem is to determine the absolute position of any joint. A joint's orientation is defined as a function of its scalars. Given a set of

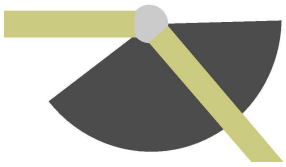


Figure 2.3: *A constraint has been set to the joint*

rigid bodies linked with joints, the posture of this kinematic model is specified by the orientation of the joints. The absolute position of a specific joint, called end effector, can then be determine as a function of the joint scalars $\theta = \theta_1, ..., \theta_n$

$$p = f(\theta) \tag{A-1}$$

Given the kinematic model as in Figure 2.4 the equation for calculating the absolute position P_3 would result in:

$$P_{3x} = L1 * \cos(\theta_1) + L2 * \cos(\theta_1 + \theta_2) \tag{A-2}$$

$$P_{3y} = L1 * \sin(\theta_1) + L2 * \sin(\theta_1 + \theta_2) \tag{A-3}$$

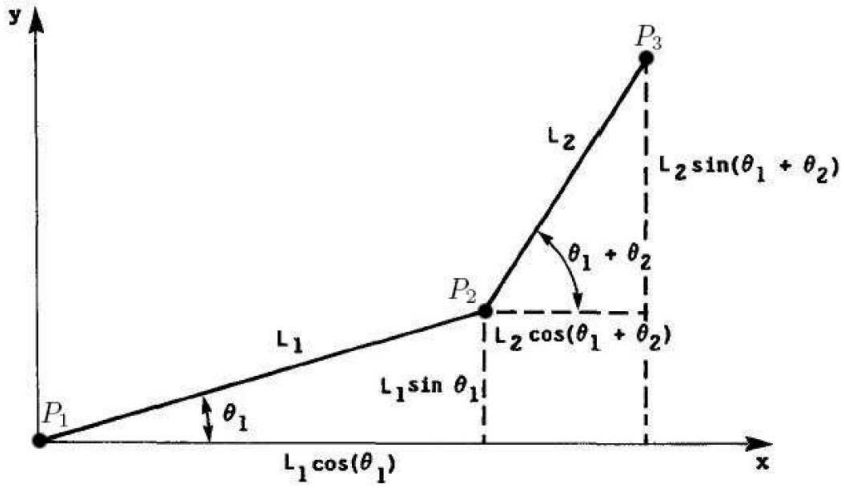


Figure 2.4: *Three joints in two dimensions*

2.3 Inverse Kinematics

The problem formulation of inverse kinematics is opposite to forward kinematics. Given equation A-1 for forward kinematics the inverse kinematics formulation

can be seen as: "Given an absolute position for an end effector, is there a configuration for the joints such that the end effector is positioned at the given absolute position?". This means from the equation A-1 a new equation can be derived as:

$$\theta = f^{-1}(p) \tag{A-4}$$

There are two distinct methods for solving the equation A-4, one is to analytically solve the equation and the other is to solve it iteratively, which means that a simplified version of the equation are calculated and evaluated repeatedly over time. Analytical solutions are precise but the complexity of it arises when large chains of joints are tried to be solved [5]. Using iterative methods good appoximations are computed with the use of the Jacobian matrix.

2.3.1 Analytical Solver

Analytical sovlers will produce all possible solutions that satisfy the given inverse kinematic problem.

If a model have enough of degrees of freedom for it to cover its workspace there is usually a finite number of solutions. If there are too few degrees of freedom there are no solutions, and if there are too many there are an infinite number of solutions. In the case of no solutions the solver would not return values that can be used for positioning the model. For example, there is no real solution to $\theta = \arccos(\frac{x}{l})$, where $x > l$. The analytical solvers basically corresponds to solving n equations with n unknows, which also is the actual problem using the analytical solvers.

The following is an example of solving an inverse kinematic problem using an analytical method that is placed in two dimensions. Consider a two segment chain as in Figure 2.4. Here P denotes a joint with a degree of freedom of one, θ the angular scalars and L the length of the segment. The relative positions of P can be written as a vector:

$$P_2 = [L_1 \cos(\theta_1), L_1 \sin(\theta_1)] \tag{A-5}$$

$$P_3 = [L_2 \cos(\theta_1 + \theta_2), L_1 \sin(\theta_1 + \theta_2)] \tag{A-6}$$

The absolute position of $P3$ can by found by adding A-5 with A-6 and a more readable form can be derived:

$$P_{3x} = [L_1 \cos(\theta_1) + L_2 \cos(\theta_1 + \theta_2)] \tag{A-7}$$

$$P_{3y} = [L_1 \sin(\theta_1) + L_2 \sin(\theta_1 + \theta_2)] \tag{A-8}$$

This can be expanded to

$$P_{3x} = (L_1 + L_2\cos(\theta_2))\cos(\theta_1) - L_2\sin(\theta_2)\sin(\theta_1) \tag{A-9}$$

$$P_{3y} = (L_1 + L_2\cos(\theta_2))\sin(\theta_1) - L_2\sin(\theta_2)\cos(\theta_1) \tag{A-10}$$

By using trigonometric formulas the equations A-9 and A-10 can be solved given an absolute positon to P_3, θ_2 and θ_1 will then be found as follows:

$$\cos(\theta_2) = \frac{(P_{3x}^2 + P_{3y}^2) - (L_1^2 + L_2^2)}{2L_1L_2} \tag{A-11}$$

$$\tan(\theta_1) = \frac{-L_2\sin(\theta_2)P_{3x} + (L_1 + L_2\cos(\theta_2))P_{3y}}{L_2\sin(\theta_2)P_{3y} + (L_1 + L_2\cos(\theta_2))P_{3x}} \tag{A-12}$$

To this specific problem there are two solutions,θ_2, θ_1 and $-\theta_2$, $\theta_1 + 2\alpha$, this example is also mention in [5].

Since these equations of these systems are in forms of sines and cosines they can be solved systematically. However a general solution for large chains of segments and joints is not feasible, because the complexity of the intermediate terms grows fast when variables are eliminated.

The positive effect of analytically solvers are the precise scalars they give as output, once a solution is reached, joints can be altered and the end effector will reach its goal with a single calulation. A downside is that when there are many solutions it is hard to know which solutions that are good. A good solution includes how the joints are positioned, and on graphical models the joints should be positioned in natural way.

2.3.2 Iterative Solver

The iterative methods uses the Jacobian matrix which is a linear approximation of a differentiable function near a given point. Similar to the analytical solver there is a function for each joint that describes instead of their relative position, the movement of the end effector. However, since both of these functions are non linear, a linear approximation is derived from the relative position such that it describes a partial movement of the end effector. From these approximations a Jacobian matrix can be created.

$$J(\theta) = \frac{\partial f}{\partial \theta} \tag{A-13}$$

Each column of J describes an approximated change of the end effectors position when changing the corresponding joint scalar. By multiplying the Jacobian matrix with a set of scalars $\Delta\theta$ which describes the a change of each scalar in the configuration, an approximate change ΔP of the position of the end effector will be resolved.

$$\Delta P = J(\theta)\Delta\theta \tag{A-14}$$

Equation A-14 results in an estimated change of the end effector which is similar to the forward kinematic problem seen in equation A-1. The difference between them is that equation A-14 is a linear approximation and easier to resolve. An approximate change of the scalars $\Delta\theta$ can be resolved by inverting the function $J(\theta)$. The equation A-14 can then be written as:

$$\Delta\theta = J(\theta)^{-1}\Delta P \tag{A-15}$$

This means in other words, given an articulated structure with a set of n scalars θ and an end effector P, a Jacobian matrix can be created by using equation A-13. By inverting the Jacobian, approximated scalars $\Delta\theta$ can be resolved with equation A-15.

Since the Jacobian matrix is created by linear approximations, the result $\Delta\theta$ from equation A-15 are also approximations of the change in the joint scalars. This leads to an iterative method since equation A-15 gives approximated changes in θ the equation needs to be done repeatedly. Let ΔP be $\vec{e} = t - P$, where t is the target position, P is the end effector position and $\vec{e}$ is the desirable change of the end effector. The first iteration will result in a θ from equation A-13 and equation A-15 which are added to the articulated model joint scalars. By using forward kinematics a new position P of the end effector is acquired and a new iteration begins. This is done until either $\vec{e}$ is small enough or the end effector does not move.

2.3.3 Creating the Jacobian

Calculating the Jacobian is usually easy following [2]. Consider a joint s_i with one degree of freedom which have a joint scalar θ_i, let $\vec{a_i}$ be a unit vector with the direction of the rotation axis of s_i, finally let $\vec{q_i}$ be the vector from the position of s_i to the position of the end effector P. A column vector in the Jacobian will then correspond to

$$\frac{\partial f}{\partial \theta_i} = \vec{a_i} \times \vec{q_i} \tag{A-16}$$

This is also visualized in Figure 2.5.

If the ith joint have a transitional degree of freedom the partial approximation of P will then be along the axis vector $\vec{a_i}$.

$$\frac{\partial f}{\partial \theta_i} = \vec{a_i} \tag{A-17}$$

This results in a $m \times n$ matrix where m is the dimension which the articulated model is in and n the number of degree of freedom.

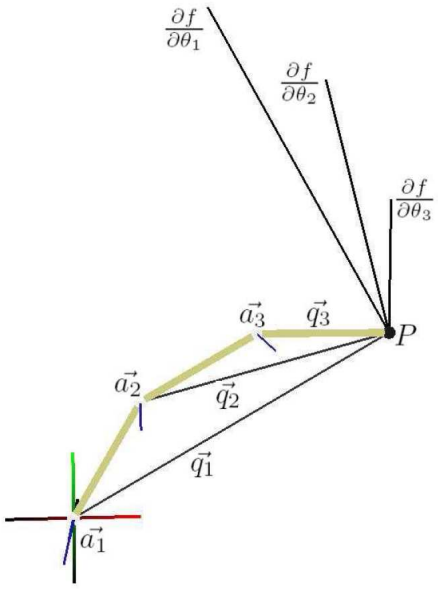


Figure 2.5: *A graphical kinematic model. The black joint marked P is the end effector, the lines point out from it are the partial derivates from equation A-16.*

There is also a more complex version of the Jacobian where $m = l \times k$, where l is the dimension and k the number of end effectors involved in the Jacobian which also will be discussed later in this thesis.

2.3.4 Using the Jacobian

All methods for solving inverse kinematics that are discussed in this thesis are using the Jacobian matrix as its basis.

There are two ways of using the Jacobian matrix to solve inverse kinematics. One is to use the transpose of the Jacobian J^T [7, 8]. The other is to calculate the inverse of the Jacobian J^{-1}, which is actually needed following the equation A-15.

Inverting the Jacobian matrix is not trivial, following the section 2.3.3 it turns out that the jacobian is non square if the degree of freedom is greater than the dimension the model is in. Alternative methods for inverting the Jacobian is then needed. Methods of solving J^{-1} are discussed in [2, 10, 6] and some methods will be discussed with more depth in this thesis.

2.4 Solvers of inverse kinematics

There are several methods of solving inverse kinematics with the Jacobian, transpose methods [7, 8], pseudoinverse methods [4], Damped Least Squares method [9], Kim and Buss' Selectively Damped Least Squares method [3, 2]. Most of these methods are also mentioned and explained in [2].

Unreachable targets are also discussed in some of the papers. Some of the methods suffers from jerky movments when the end effector is trying to track a position that is out of range. In section 2.4.3 one way of tracking a position is presented. Another way to reduce this is by clamping the range to the target positions introduced in [3]. What they do is to instead of just setting $\vec{e}$ as the desirable change, a method reduces $\vec{e}$ if it is too large.

Jerky movements is a result of large irregular changes of the scalars θ. These large changes comes when the Jacobian matrix is about to loose full row rank and when the target position is out of range. In other words when the model as in Figure 2.6 is almost fully extended, the partial derivates are all almost along the same line (two dimensions) or plane (three dimensions). Solving equation A-15 or in other words constructing $\vec{e}$ from the partial derivates will then result in large scalars, therefore the jerky movements.

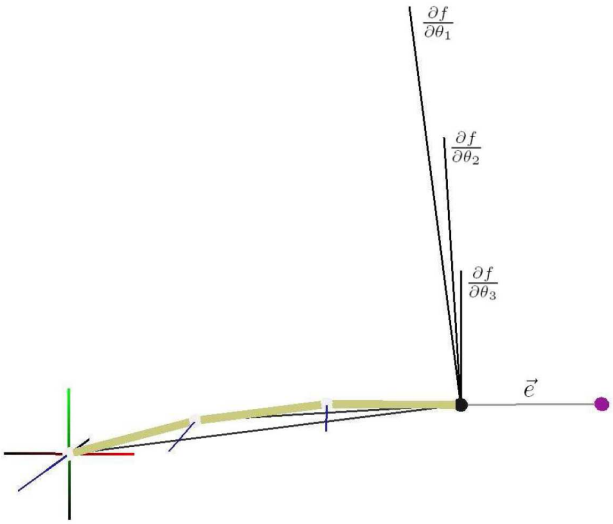


Figure 2.6: *A graphical kinematic model in an almost exended posture.*

Joint constraints is not a part of the Jacobian matrix, instead once a change in the angular scalar is received, the constraint decide whether it is possible to change the angle. When an articulated model has a large number of constraints it will then make the solver slower to reach its target position.

2.4.1 Transpose

This method have been used for inverse kinematics among other by [7]. The method uses the transpose of the Jacobian matrix instead of the inverse. It has been proven in [2] that it is possible to justify the use of it, simplified they say that the dot product of $\frac{\partial f}{\partial \theta_i}^T$ and $\vec{e}$ will result in a scalar that have the correct sign but the magnitude is of wrong size. This method resolves the equation A-15 to

$$\Delta\theta = \alpha J^T \vec{e} \tag{A-18}$$

for some scalar α. A way of choosing α is also presented in that paper. They use the relationship between the desired change $\vec{e}$ and an approximated change of the end effector which is defined as $JJ^T\vec{e}$. By assuming that $JJ^T\vec{e}$ is the real change, α is chosen such that it is as close as possible to $\vec{e}$:

$$\alpha = \frac{< \vec{e}, JJ^T\vec{e} >}{< JJ^T\vec{e}, JJ^T\vec{e} >} \tag{A-19}$$

This method is according to [3] fast to calculate since it do not need any solving of inverses.

2.4.2 Pseudoinverse

The pseudoinverse method is used to solve equation A-15. It is used because J is most likely redundant and non square, thus an ordinary inverse is not possible. The resulting scalars are used for rotating the joints degrees of freedom. Pseudoinverse is also called Moore-Penrose inverse. This method sets $\Delta\theta$ equal to

$$\Delta\theta = J^\dagger \vec{e} \tag{A-20}$$

where $J^\dagger$ is the pseudoinverse of J. $J^\dagger$ is defined for matrices in [11]. Properties of $J^\dagger$ which are also shared with normal inverses are:

$$JJ^\dagger J = J \tag{A-21}$$

$$J^\dagger JJ^\dagger = J^\dagger \tag{A-22}$$

$$\left(J^\dagger J\right)^* = J^\dagger J \tag{A-23}$$

$$\left(JJ^\dagger\right)^* = JJ^\dagger. \tag{A-24}$$

Where * is the conjugate transpose, if J contains all real values J^* is equal to J^T. Note that $J^\dagger J$ is a symmetric square matrix also known as Hermitian.

In [2] they state that using the pseudoinverse will give the best solution to the equation $J\Delta\theta = \vec{e}$ in the sense of least squares. If $\vec{e}$ is in the column span of J, $\Delta\theta$ is an unique vector of smallest magnitude satisfying A-13. If $\vec{e}$ is not in the column span an exact value of $\Delta\theta$ is impossible to get, however $\Delta\theta$ minimizes the magnitude difference of $J\Delta\theta - \vec{e}$.

Problems of the pseudoinverse method is that it has stability problems near singularity. These problems comes when the Jacobian matrix is about to loose full row rank which arises when the model is in an extended posture seen in Figure 2.6. In terms of model behavior it results in large angular changes such that the model moves without control, this is also known as jerky movements explained in the previous section 2.4. Keep in mind that each column vector of J are only approximations. Which means that they only corresponds to the tangential movement and not the real movement of the end effector when changing their degree of freedom. Because of that fact, solving $\Delta\theta$ with A-15 can result in scalars which does not move the model towards its target.

In a real singularity the method will not attempt to move the end effector towards the target position. If the model in Figure 2.6 had all its joint extended such that the partial derivates are aligned along the same vector, the Jacobian matrix would be singular. But due to roundoff errors in practice a real singularity will rarely occur.

The algorithm for the pseudoinverse method is derived from A-13 as the normal equation

$$J^T J\Delta\theta = J^T\vec{e} \tag{A-25}$$

solving $\Delta\theta$ A-25 can be rewritten to

$$\Delta\theta = (J^T J)^{-1}J^T\vec{e} \tag{A-26}$$

$J^T J$ will produce an $n \times n$ size matrix, where n is the number of degrees of freedom in the model, and the inverse of it can be expensive to calculate. In [11] it is shown that if $J^T J$ is invertible the following is true:

$$J^\dagger = (J^T J)^{-1}J^T \tag{A-27}$$

and if JJ^T is invertible:

$$J^\dagger = J^T(JJ^T)^{-1} \tag{A-28}$$

the equation A-26 can be rewritten

$$\Delta\theta = J^T(JJ^T)^{-1}\vec{e} \tag{A-29}$$

that results in an inverse of an $m \times m$ matrix, where m is the dimension the joints is in. Which mostly is less than n. Additionally the inverse in equation A-29 do not need to be computed analytically. By solving $\vec{\omega}$ in $JJ^T\vec{\omega} = \vec{e}$ using Gaussian elimination, then $J^T\vec{\omega} = \Delta\theta$. However if J is not full row rank a single solution will not exist.

An other property of the pseudoinverse is that the matrix $I - J^\dagger J$ is a projection of J onto nullspace. Such that for any vector $\vec{\varphi}$ that satisfy $J\left(I - J^\dagger J\right)\vec{\varphi} = 0$, $\Delta\theta$ can be set by

$$\Delta\theta = J^\dagger\vec{e} + \left(I - J^\dagger J\right)\vec{\varphi} \tag{A-30}$$

In this manner $\Delta\theta$ will still result in a value that minimizes $J\Delta\theta - \vec{e}$. In [4] the nullspace is used to determine the error of the calculated pseudoinverese.

$$Error = ||(I - JJ^{\dagger})\vec{e}|| \tag{A-31}$$

Since $JJ^{\dagger}\vec{e}$ gives an approximated change of the end effector, hence equation A-31 will give a magnitude error of the desired change. If the error is too large, $\vec{e}$ is reduced to half it length.

2.4.3 Damped Least Squares

Using the damped least squares is a way of removing and reducing near singularites in the Jacobian matrix and stabilizing $\Delta\theta$. This method is also called Levenberg-Marquardt algorithm and one of the first to use it was Wampler [9]. In equation A-14 the search for a minimum vector $\Delta\theta$ that gives the best solution was wanted. In this method, $\Delta\theta$ that minimizes

$$||J\Delta\theta - \vec{e}||^2 - \lambda^2||\Delta\theta||^2 \tag{A-32}$$

where λ is a non zero real value constant. Equation A-32 can be rewritten as

$$\left|\left|\begin{pmatrix} J^T \\ \lambda I \end{pmatrix}\Delta\theta - \begin{pmatrix} \vec{e} \\ 0 \end{pmatrix}\right|\right| \tag{A-33}$$

such that the normal equation A-25 resolves to

$$\begin{pmatrix} J \\ \lambda I \end{pmatrix}^T \begin{pmatrix} J \\ \lambda I \end{pmatrix}\Delta\theta = \begin{pmatrix} J \\ \lambda I \end{pmatrix}^T \begin{pmatrix} \vec{e} \\ 0 \end{pmatrix} \tag{A-34}$$

Following the steps in equation A-27, A-28 and A-29 the solution to $\Delta\theta$ will look like

$$\Delta\theta = J^T(JJ^T + \lambda^2 I)^{-1}\vec{e} \tag{A-35}$$

What this means is that by adding the orthonormal set I it is guaranteed that an inverse exists, however choosing λ too small will still result in jerky movments, when the model is in an extended posture, which is the same problem as with the pseudoinverse method. This means in mathematical terms that by letting λ in equation A-35 approach zero it results in equation A-29. If λ is too large, the scalars $\Delta\theta$ will become smaller such that each iteration will only change the end effector position very little and the number of needed iteration to reach the goal would grow.

The constant λ must be chosen such that neither of the above becomes a problem. It is suggested that the constant depends on details of the articulated model, but there is no mention of selecting the constant directly in [2], however it have some references to a number of authors that proposes a way of selecting it dynamically.

A more practical way of explaining this method is by using Figure 2.6 where the model is in an extended posture and its corresponding Jacobian matrix is in near singular. Solving the Jacobian would normally result in large scalars and in its turn a jerky model behaviour. By adding two vectors orthogonal to each other, along x-axis and along y-axis, in the Jacobian matrix, solving $\vec{e}$ becomes easier and would result in smaller scalars. The constant λ determines how long the two vectors are, and by choosing a large λ the two vectors relieves the scalars from resulting in large values.

2.4.4 Singular Value Decomposition

Singular value decomposition is not a method in it self, but it can however be used to calculate and analyze the pseudoinverse of the Jacobian. It is used by [3] to optimize the damped least squares method to their selectively damped least squares method.

It is stated in [10] that if J is an $m \times n$ matrix with row rank r it can be decomposed to

$$J = UEV^T \tag{A-36}$$

where U is a $m \times m$, E a $m \times n$ and V a $n \times n$ matrix. E contains diagonal entires of the r first singular values of J,$\sigma_i = e_{i,i}$,$\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_r$. U and V are orthogonal matrices. Where U forms an orthonormal basis for $\Re^m$ and V for $\Re^n$ which also is known as the nullspace of J. The pseudoinverse of J with SVD is defined by

$$J^\dagger = VE^\dagger U^T \tag{A-37}$$

Since E contains a diagonal $r \times r$ matrix D, E is formed like

$$E = \begin{pmatrix} D & 0 \\ 0 & 0 \end{pmatrix} \tag{A-38}$$

and the pseudoinverse

$$E^\dagger = \begin{pmatrix} D^\dagger & 0 \\ 0 & 0 \end{pmatrix}. \tag{A-39}$$

The inverse of D is defined by

$$d^\dagger_{i,i} = \begin{cases} 1/d_{i,i} & if d_{i,i} \neq 0 \\ 0 & if d_{i,i} = 0 \end{cases} \tag{A-40}$$

It is also common to write A-36 as a sum of vectors

$$J = \sum_{i=1}^{m} \sigma_i u_i v_i^T = \sum_{i=1}^{r} \sigma_i u_i v_i^T \tag{A-41}$$

then the pseudoinverse is equal to

$$J = \sum_{i=1}^{r} \frac{1}{\sigma_i} v_i u_i^T \tag{A-42}$$

The properties that can be exploited in SVD lies withing the matrix E and its singular values which have been used by [3].

2.4.5 Full and Half Jacobian

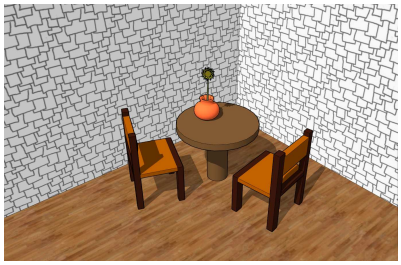
In [4] they introduce the concept of full and half jacobian, where half means positioning and full positioning and orientation. In case of positioning the desired change $\vec{e}$ only consider the position, $\vec{e} = [\Delta x, \Delta y, \Delta z]$ while the full also include the current orientation of the end effector, such that $\vec{e}$ is a vector containing six elements. The full Jacobian adds additional information to each entry, the first three rows contains the partial derivates from each degree of freedom, the last three contains information of the rotation axises of the degrees of freedom.

2.5 Scene Graph

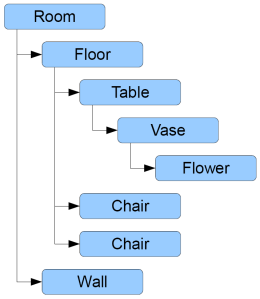
A scene graph is a structure which is often used to arrange the logical and spatial representation of a graphical scene. There is not a clear definition of a scene graph since programmers who implement them generally only use the basic principles to create structure suiting the application [14].

The basic principles of a scene graph is a collection of nodes arranged into a graph or tree structure. This indirectly implies that a node may have many children but seldom more than a single parent. If an operation is applied to a node the effect propagates to all its members. In many cases such operation consists of a transformation matrix which is associated with the node. The matrices are easy to concatenate and it can be done efficently when the operation propagates down to its children [13].

The transformation matrix associated with a node is generally a 4×4 matrix formed by concatenating one or several geometrical transformation matrices which can among other describe rotations, translations and scaling. In other words the transformation matrix describes the location and orientation of a node's coordinate system relative to its parent.



(a) *A 3D scene representing a room.*



(b) *The scene graph representation of the room.*

Figure 2.7: Scene graph

Chapter 3

Implementation

3.1 COLLADA

COLLADA[1] is short term for **COLLA**borative **D**esign **A**ctivity which defines an extensible markup language (XML) scheme that enables free exchanges of digital assets for 3D-applications without loss of information. COLLADA was initiated by Sony computer entertainment to create a standard for digital assets and a goal to liberate it from proprietary binary formats and was further developed by the Khronos group[2] in 2006. The Khronos group was founded in 2000 by leading media centric companies, including 3Dlabs, NVIDIA, SGI, Sun Microsystems, Intel, ATI, SGI and Discreet. The group is dedicated to create open standards to a variety of devices and platforms.

There were several goals and guidelines for the COLLADA digital asset schema when it was created [12]. One of them "To liberate digital assets from proprietary binary formats into a well-specified, XML-bases, open-source format". This goal was set because a large part of a 3D-application is defined by digital assets. Investments on proprietary binary formats are considerably large, and even after these investments it only gives the developer the ability to export and import the format, not a possibility to modify the data outside the tool and import it later.

This goal led into a few decisions regarding the format of COLLADA, including:

- It will use XML. XML provides a well defined standard that deals with issues such as character sets, and parsers for almost all languages and platforms already exists.
- No binary data will reside within the COLLADA file. Since many lan-

[1] http://www.collada.org/
[2] http://www.khronos.org/

guages do not easily support binary data within XML and in general they do not have support for manipulation, so by having no binary data it simplifies it. COLLADA does however provide means to reference an external binary file.

These decisions lead to a COLLADA document with some rules. A COLLADA document contains a mandatory header element, while libraries and a set of scenes are optional, which all are set under the root element <COLLADA>. The header element is tagged with <asset> and containts basic information about COLLADA-version used, author and other information that the developer consider as individual assets.

3.1.1 COLLADA Libraries

A COLLADA document is built from a sequence of organized libraries of specific types. A library describes some part of a scene, for example an animation. The libraries that are used in this thesis are the animation library and the visual scene library.

3.1.2 Animation Library

The animation library contains a set of one or more animation elements. Each animation element basically describes a transformation of an object or value over a specified time period, most commonly to give a model the illusion of motion. A form of animation is the simple key-frame animation where a key-frame contains a value that is linked with a key. Given a set of these key-frames an animation can be derived. In other words, given a model and a set of key-frames, where the keys are time indexes and the linked values are positions of a models arm, an interpolated postion of the model's arm can be calculated given any specified time within the key-frames [12].

To make the previous possible a COLLADA-animation consist of three elments: a source, a sampler and a channel element. The source element <source> is a reference to a one dimensional array that can be accessed. In order to create a key-frame two sources are needed, one "input" where the time indexes are stored and one "output" that will contain values corresponding to the time indexes. Interpolation type can also be set in the source element, other elements within this have information including length of the array and stride. The stride tells the sampler element how many of the values belongs together.

The <sampler> element is used as an intermediator to sort out what values in the source element that is input and output and to create n-dimensional arrays from the values.

The <channel> element connects the output values to the object that shall be transformed. It has two mandatory elements: a reference to a source and

a target. In this case the source would be the output from the sampler, and target is the object.

3.1.3 Visual Scene Library

The visual scene library contains a set of one or more visual scene elements. A visual scene element describes visual aspects of a COLLADA document. The element forms a root of an organized hierarchic structure formed like a tree data structure, also known as a scene graph, of <node> elements that contains visual information and related data.

The <node> element has five optional attributes:

- *id* - A text string containing the unique identifier of the element. This value must be unique within the instance document.
- *name* - The text string name of the element.
- *sid* - A text string value containing the subidentifier of this element. This value must be unique within the scope of the parent element.
- *type* - The type of the <node> element. Valid values are JOINT or NODE. The default is "NODE".
- *layer* - The names of the layers to which this node belongs.

The *type* attribute is the main attribute that will be used within this thesis. If that attribute is set to the value JOINT it means the node is part of an articulated structure for example a kinematic model.

A node can also contain instances of cameras, controllers, geometries and lights where the instances are defined in their respective library. They can also contain other nodes which is seen as children. Nodes defines their own local coordinate system which is relative to their parent. Their relative coordinate system is decided by any number of three dimensional tranformations. COLLADA defines six transformation elements that may be used within these nodes: *lookat*, *matrix*, *rotate*, *scale*, *skew* and *translate*, where all can be transformed into a 4×4 matrix for accumulation. By altering the transformations a desirable position or orientation can be attained.

3.2 COLLADA and IK

COLLADA version 1.4 does not have a specific implementation of inverse kinematics and since AgentFX is tightly connected to the COLLADA-structure, it was not an option to fully integrate this application within AgentFX. However, using attributes set by the standard it is possible to retreive information from

the COLLADA document to build a kinematic model to an application which will be made for the purpose of this thesis.

The data structure of the kinematic model will mimic the scene graph in the COLLADA document in the sense of parent and child relationship and their geometrical transformations. An extra geometrical transformation matrix will be added in the structure to easier be able to separate the movement made by the inverse kinematic algorithm from the state the model was in at start.

Each joint in the model will be given six degrees of freedom, three rotational and three transitional, which could be turned on or off. If turned off the algorithm computing the inverse kinematic will not consider that particular degree of freedom.

3.3 Deriving the Jacobian

Two methods of deriving a Jacobian matrix were implemented to find out the difference between them. In [2] they create a Jacobian matrix with the size of $m \times n$ where $m = 3k$ where k is the number of end effectors and n is the number of degrees of freedom affecting the end effectors. Another way is to create k Jacobian matrices, one for each end effector, and now $m = 3$.

Following the method in 2.4.2 a Gaussian elimination will be needed to solve $\vec{\omega}$ in $(JJ^T)\vec{\omega} = \vec{e}$. Gaussian elimination have a complexity of $O(n^3)$ where $n = 3k$ using the first metod and $n = 3$ using the second method.

The first method is good because it only give one solution for each iteration and a single update of the joint scalars are performed. Imagine a hand where the fingertips are the end effectors and the wrist the root, which also is the only common joint for the end effectors. This method creates a Jacobian matrix in the following way. Consider a set of k end effectors, $P = \{P_1, \ldots, P_k\}$, and a set of joints with a total of n scalars, $\mu = \{\theta_1, \ldots, \theta_n\}$, which describes the joints configuration. Then the positions of P is described as

$$P_i = f_i(\theta) \quad \text{for } i = 1, \ldots, k. \tag{A-1}$$

The Jacobian is then created in the same way as in 2.3.3 with the addition, if the ith end effector is not affected by the jth, $j = 1, \ldots, n$, joint, the partial derivate is set to zero. This creates a matrix of the size $3k \times n$.

The second method creates a Jacobian with only necessary entries. In a sense it divides the model into submodels. A submodel contains an end effector, a root and all joints between them in the hierarchy. The submodel's Jacobian matrix will therefore always have the size $3 \times n$. A submodel can share joints between other submodels, as with the example with the hand all submodels share the wrist as a common joint. Altering a common joint will, due to the parent-child realtionship, change the orientation of all the submodels sharing that joint.

Each iteration consists of creating the Jacobian, solving the Jacobian and finally update the submodel's joints with the given changes, this will be done for all submodels. A side effect is that it will result in multiple updates on all common joints in each iteration when the model have multiple end effectors. In other words, given the same hand as above there are five submodels, one for each fingertip. A Jacobian matrix would first be created with the first finger as an end effector and the wrist as a root. Then by solving the first matrix, by any mentioned method, a set of joint scalars can then be applied to the corresponding joints. This step will be repeated for all submodels. By using this method the wrist joint itself would be changed five times each iteration. In a really bad scenario the first submodel could for instance change the wrist $90°$ and the second submodel change it back $90°$, this results in that the first change to the shared joint would be more or less gone. In addition to that, the first submodel's end effector would be further off its target than it was before due to the parent-child relationship.

3.4 Mathematic Tools

Several tools were needed to be able to compute a scene graph and calculations related to the creation of the Jacobian matrix. Most of the mathematic methods involving linear algebra and computing 3D content was found within AgentFX that were ready to use. However a method for solving $Ax = b$ using Gaussian elimination was needed, as mention in a previous chapter a part of the pseudoinverse do not need the inverse that is seen in the methods.

3.5 Animation

AgentFX provides the means of loading and saving a COLLADA document. By loading the document it creates a scene graph where it is possible to modify and add extra content. The content that this application shall be able to add is animation information. A method that adds a key-frame for all joints that could be called upon was needed.

Chapter 4

Evaluation

The following methods were tested to solve inverse kinematics: transpose, modified transpose, pseudoinverse, pseudoinverse with null space control and damped least squares.

4.1 Constants

The constants chosen in the methods are described below. The dampening constant λ for the damped least squares method were not present in any of the referenced papers. Because of this a small study of the behavior for the damped least squares method were used to determine a balanced dampening constant which considered the current state of the articulated model at each iteration.

The resulting column vector $\Delta\theta$ from all the methods were multiplied with a scalar set to 10, otherwise the articulated model converged to the target positions too slowly.

The following section describes how the constants were chosen in their method. The pure pseudoinverse method did not have any constants to be set.

4.1.1 Transpose

The constant α was chosen in the same way as described in [2]

4.1.2 Modified Transpose

The difference in this modifed transpose method is a dampening factor much like the one described in damped least squares, which is added into the Jacobian matrix in order to stabilize when tracking targets out of range. The constant α was chosen like in the transpose method. The dampening factor was set dynamically as the magnitude of the desirable change added with the magnitude of the distance between the end effector and the root.

4.1.3 Damped Least Square

The dampening constant λ was chosen as the magnitude of the desirable change of the end effector added with a small constant which was set at each iteration. The small constant was in this case 1/100 of the total sum of the segments from root to end effector. The magnitude of the desirable change is a nice value of λ since it more or less ensures that the Jacobian matrix does not get near singular at any time. The small constant were added because when the target positions were nearly out of range, it started oscillating since the desirable change was too small to dampen.

As seen in Figure 2.6 an orthogonal set of vectors where each vector has the length of the desirable change vector $\vec{e}$ will damp effciently. This will however also reduce the speed when setting target positions that are in range but far from the end effector.

4.1.4 Pseudoinverse with null space

This method was derived from [4] and is also explained in the pseudoinverse chapter. The maximum error was set to 1.

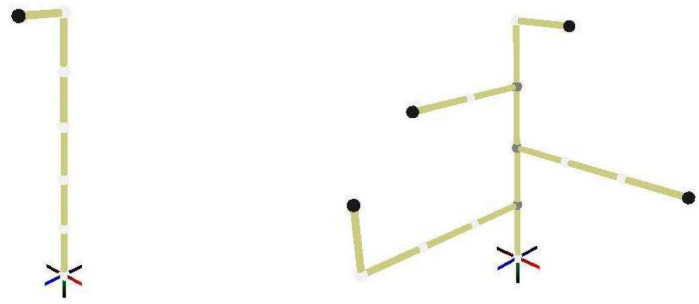
4.1.5 Constants linear relationship

Since some constant values in the methods were chosen and set at each iteration it was vital to see if they had a linear relationship. In order to determine if they are correctly chosen the model and the target positions were resized with a scalar. By first comparing the number of iterations by the first model and paths with the resized model, they should remain the same. It turned out that the constants were chosen with a linear relationship, resizing the model and the target positions with the same scalar did not affect the number of iterations.

4.2 Models

There were two different models that were used, one model had a single end effector and 18 degrees of freedom Figure 4.1(a) and a model with four end effectors and a total of 33 degrees of freedom Figure 4.1(b). Additionally a model with 180 degrees of freedom were also used to determine if the methods were stable in a large chain. The range between each joint was set to 1 unit.

From the problem description two different tests were derived from each configuration to test the robustness. One test the target positions were in range for the end effector, the other test the targets where out of range. The target positions (path) for the single arm had nine targets and the multiple arm configuration had four target positions. The second configuration with four end effectors were tested with two different methods, the first was with



(a) *Test model with one end effector and 18 degree of freedom.*

(b) *Test model with four end effectors and 33 degrees of freedom.*

Figure 4.1: Test Models

four Jacobians calculated separately, and the other was with the method that created a single Jacobian out of multiple end effectors (see 3.3).

When the path was set to in range, the end effector had to be within 0.01 units of the target position to be considered as done. When the path was set to out of range, the end effector had to change less than 0.001 units between each iteration to be considered as done. Note that when the path was set out of range, the end effector did not need to be near the target position to be considered as done, all it checked for was the movement of the end effector.

After it was considered as done, next target position in the path was iterated. If the number of iterations exceeded 1000 they were considered as not reaching the goal, and the next target position was set.

As stated in the problem description the inverse kinematics also needs to be fast enough such that current computers do not suffer when using these methods. The tests then also included the mean number of iterations for a method to converge to a target position and the mean time it used to calculate a single iteration.

An additional model was also tested which had one end effector and 180 DOF. It had nine in range target positions and were run for 100 cycles. This was made to control the robustness in large structures.

4.3 Tests

The tests were performed on a AMD Athlon XP 2200+ 1.80 GHz with 1 GB of RAM, GeForce 6600 LE graphic card, run on Windows XP, the methods were implemented with JAVA SE 6.0. Each configuration and test were run 1000 cycles, where a cycle is complete run of a set of target positions. Transpose' is the modified transpose method and Pseudoinverse' is the pseudoinverse with null space method.

The column "time for one iteration" in the presented tables is the total time it takes to create Jacobian, calculating $\Delta\theta$ and updating the inverse kinematics tree structure. Creating Jacobian and the update are the same for all methods, however the total time is of interest when there is a need to find out how many of these iterations can be done within each update of the graphical environment.

Method	Mean iterations	Time for one iteration
Pseudoinverse	32	96.4 μs
Pseudoinverse'	46	104.1 μs
DLS	35	89.2 μs
Transpose	59	83.3 μs
Transpose'	163	84.8 μs

Table 4.1: Model with one end effector, 18 DOF, target positions in range

Table 4.1 shows that the Pseudoinverse method have the least number of iterations, the time for one iteration is a bit higher due to the use of an analytical inverse of the Jacobian matrix, while the DLS are solving with row reduction. However Pseudoinverse' needs an analytical inverse to calculate the error magnitude, its even higher calculation time comes from the control of the error.

Method	Mean iterations	Time for one iteration
Pseudoinverse	-	- μs
Pseudoinverse'	-	- μs
DLS	74	81.5 μs
Transpose	117	73.8 μs
Transpose'	98	77.8 μs

Table 4.2: Model with one end effector, 18 DOF, target positions out of range

Table 4.2 shows that both pseudoinverse methods could not produce any result when target positions were out of range. The pure pseudoinverse method gets unstable (see 2.4.2) and the joint scalars got out of hand. On the other hand the pseudoinverse' did not get unstable, instead the error control and in some degree the implementation of it made this method unable to move once reaching for the first target position was done. When the next target position should be reached for the end effector did not move more than 0.001 units and the method thought it was done.

Table 4.3 shows when having a model with 180 DOF the pure pseudoinverse method got a bit unstable after a few cycles and the number of iterations increased in comparison to DLS.

Method	Mean iterations	Time for one iteration
Pseudoinverse	59	684 μs
Pseudoinverse'	343	715 μs
DLS	45	593 μs
Transpose	130	580 μs
Transpose'	153	586 μs

Table 4.3: Model with one end effector, 180 DOF, target positions in range

Method	Mean iterations	Time for one iteration
Pseudoinverse	165	552 μs
Pseudoinverse'	169	572 μs
DLS	168	520 μs
Transpose	388	496 μs
Transpose'	385	504 μs

Table 4.4: Model with four end effectors and four separated Jacobians, 33 DOF, target positions in range

When using four Jacobian matrices and the target positions was in range, all the methods except the transpose methods performed more or less equally which can be seen in Table 4.4. Each Jacobian have the size of 3×15 and all of the end effectors have two or more common joints, which results in these number of iterations.

Method	Mean iterations	Time for one iteration
Pseudoinverse	31	325 μs
Pseudoinverse'	120	568 μs
DLS	55	339 μs
Transpose	472	308 μs
Transpose'	-	- μs

Table 4.5: Model with four end effectors and a single Jacobian, 33 DOF, target positions in range

By merging the four Jacobians into a single Jacobian both mean iterations and the time for calculating a single iteration decreased considerably for the pseudoinverse methods and DLS, as seen in Table 4.5. The transpose methods suffered from this and gave bad results, the pure transpose did not reach the goal 433 times, about 10%. The modified transpose method performed so bad that it did not reach the goal 95% of the time. A visual study showed that even though the end effectors were fast to move fairly close to the target positions,

they converged too slow to achieve the goal within 1000 iteration.

Method	Mean iterations	Time for one iteration
Pseudoinverse	-	- μs
Pseudoinverse'	-	- μs
DLS	390	476 μs
Transpose	-	- μs
Transpose'	350	452 μs

Table 4.6: Model with four end effectors and four separated Jacobians, 33 DOF, target positions is out of range

Table 4.6 shows that the out of range test when using four Jacobians also had the same effect as the single out of range test with the pseudoinverse methods. The pure transpose method oscillated when a visual study was made to investigate the loss of results. The DLS method was stable and the modified transpose method converged to its target the fastest.

Method	Mean iterations	Time for one iteration
Pseudoinverse	-	- μs
Pseudoinverse'	-	- μs
DLS	372	333 μs
Transpose	284	305 μs
Transpose'	440	318 μs

Table 4.7: Model with four end effectors and a single Jacobian, 33 DOF, target positions is out of range

Table 4.7 shows that the pure Transpose method was fast when using the single Jacobian. The Transpose' method in this case there were 28 times the method could not solve within 1000 iterations.

4.4 Summary

The only method to produce results in all the tests in the previous section were the Damped Least Squares method, this due to the dampening factor that effectively stabilizes the resulting joint scalars when the target positions were out of range. The Damped Least Squares method is also a fairly fast method and is often placed right after the fastest method when counting the number of iterations.

The Pseudoinverse method reaches the target positions with the least number of iterations when they were in range. But when the target positions are out

of range, the lack of a dampening factor causes the Jacobian matrix to get near singular and the resulting joint scalars became too large such that the model started to move irradically.

The Pseudoinverse method which were using null space to try stabilize the joint scalars when target positions were out of range needed most time for each iteration. This method did not produce any results when the target positions were out of range which part can be blamed on the implementation and the other part can be blamed on the method it self.

The transpose method produced results in all tests except when it used four Jacobians and the targets were out of range. The modified transpose method only outperformed the other methods when using multiple Jacobians with multiple end effectors. When using a merged single Jacobian the number of iterations and time generally decreased.

Chapter 5

Conclusions

This thesis has presented test-results of how different methods of inverse kinematics perform in various situations in order to determine if they are suitable to be used in real-time applications. An application for testing the methods were also built.

If I were to use any method to solve the inverse kinematic problem I would use the damped least square method. Looking at the results it can handle both target positions that are in range and out of range without generating any jerky movement to the model. One problem with the damped least square method is the way I chose the constant λ. Since the constant depends on the length of the desired change, the farther away the target position was compared to the end effector the slower the method would converge. In [2] they explain that a way of removing this side effect is to clamp the desired change. Given a model without transitional joints it can move at most two times the total length of the model (the sum of all segment lengths from the end effector to the root. Clamping the desirable change to that length will result in a faster method when the distance from the end effector to the target position are farther than the total length of the model.

The transpose method is supposed to be a fast algorithm according to [3]. However, looking at the results, there is nothing that indicates that. I tried to make it faster by increasing the scalars output with a larger constant but it only made it more instable.

My attempt to make the transpose method more stable worked in the case where there were several Jacobian matrices involved and the result surprised me, but with the single Jacobian it was a bad choice.

Is it then possible to use this to create an animation by directing the end effectors and take key frames in real time? The answer is yes, given the current computers it is possible. Lets say that you have a 3D game that would give the inverese kinematic method 10 ms to iterate a solution each frame, which means

that the game would be able to reach up to 100 frames per second. If the model had the configuration with the multiple end effectors, and the same computer these methods were tested on. The computer can iterate about 29 times. By using the DLS method, half the number of iterations can be accomplished in that time needed for the model to reach its target when the target position is in range. But since an animation rarely moves from one posture into another instantly, it is not a problem that the model only made it half way.

Due to the lack of literature explaining how constraints should be handled in a way where the inverese kinematic methods don't give results that can exceed the constraints. What you would want is a method that doesn't give results that can exceed the working limits of the joint. Therefore, any documented tests for handling constraints were never done in this thesis. However, in [6] a very simple way of implementing constraints are presented, which checks all the joints after each iteration. If a joint has exceeded any of the constraints it forces the joint back into a position were the constraints have not been surpassed. This also means that the restrictions are never considered when creating the Jacobian matrix, all joints are treated equal. Play with the idea of a model that had a constraint set to have a working limit of zero, such as it basically is not a degree of freedom. The creation of the Jacobian matrix will however not recognize this, and treat this as any other joint. The solvers themselves doesn't either recognize constraints, they only return a summary of all changes of the joints that should move the end effector to its target. By having a joint that exceeded its constraint would then remove a part of that summary and move the end effector less close to the target. Using the simple way of dealing with constraints results in a chance of reducing the effectiveness of the solvers when dealing with a model that have any number of constraints.

5.1 Future work

There is a lot of future work on this subject and the implementation of the kinematic tree structure could be optimized. Reading [13] I saw that there can be a lot of improvement of the kinematic scene graph that is currently used. As it is now mainly 4×4 transformation matrices are used within the nodes. These are geometrical transformation matrices and they inherit some features that can be used to reduce the complexity of multiplying two matrices. I strongly believe that my implementation of the kinematic scene graph could be improved such that it only would need half the time it needs now to update its data structure.

Although this thesis did not cover much of how to constrain joints, I did play with the idea of how to improve the method mentioned in [6]. One way of improving it would be by removing a joint's degree of freedom once it has exeeded its constraint, this would result in a Jacobian matrix that does not consider that

specific degree of freedom. Solving it with any of the methods results in scalars that would be more correct and they would not loose effectiveness when more joints exceeds its constraint. However it is hard to know when a joint's degree of freedom have returned into its working limits.

An other idea I had is to add a priority scalar, φ, ranging from 0 to 1 infront of the partial derivate, $\varphi \frac{\partial f}{\partial \theta_i}$. If a degree of freedom tries to exceed its limit the priority would be reduced. This means that the partial derivate becomes smaller and the inverse kinematic method returns a smaller scalar change in that specific DOF. Once it makes a change back into its constraint the priority resets to normal.

Bibliography

[1] Encarta Dictionary, "http://encarta.msn.com", 2009

[2] Samuel R. Buss, "Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares Methods", April 17, 2004 Unpublished

[3] Samuel R. Buss and Jin-Su Kim, "Selectively Damped Least Squares for Inverse Kinemetaics", 2004

[4] Michael Meredith and Steve Maddoc, "Real-Time Inverse Kinematics: The Return of the Jacobian"

[5] BertHold K. P. Horn, "Kinematics, Statics and Dynamics of Two-dimensional Manipulators" in Artifical Inteliigence: An MIT Perspective Vol. 2, Winston, P.H. and R.H. Brown (Eds.), pp. 273-310, MIT Press

[6] Kang Teresa Ge, "Solving Inverse Kinematics constraint Problems for Highly Articulated Models",2000

[7] W.A Wolovich and H. Elliot, "A Computational Technique For Inverse Kinematics", 1984

[8] Daniel E. Whitney, "Resolved Motion Rate Control of Manipulatros and Human Prostheses"; 1969

[9] Charles W. Wampler, "Manipulator Inverse Kinematics Solutions Based on Vector Formulations and Damped Least-Squares Methods", 1986

[10] David C. Lay, "Linear algebra and its applications" second edition, 2000

[11] Lennard Råde and Bertil Westergren, "Mathematics Handbook for Science and Engineering", 1989, 1998 fourth edition

[12] COLLADA - Digital Asset Schema Release 1.4.1 (2006). http://www.khronos.org/¯les/collada spec 1 4.pdf

[13] David H. Eberly, "3D Game Engine Design", 1999

[14] Johan Lindberg, "Implementation of a COLLADA scene-graph", 2006