Master Thesis

Inverse Kinematics Techniques in the BirthPlay Application

Game and Media Technology

Author: K. van der Lei
Advisor: dr. ir. J. Egges
Thesis Number: ICA-3727785

December 15th, 2013

1 Abstract

This research focuses on the manipulation of a virtual character using real-time movement of 6-degree-of-freedom controllers. To this end, several Jacobian-based IK methods were implemented, which use an approximation of the inverse of the Jacobian matrix to calculate changes in DOF values that will lead to the desired position and orientation of the end effector (the last link in the joint chain representing the arm). These methods include the Jacobian Tranpose, Jacobian Pseudoinverse, and Damped Least Squares. Joint limits are enforced through clamping of orientation, as well as the temporary removal of some of the joint's degrees of freedom. Collisions are handled within the Jacobian matrix itself by adding additional effectors for each contact, with a correctional impulse counter to the contact's direction and magnitude based on contact depth. A weight-based approach is used to handle priority of the constraints, those being the end effector's position, its rotation, and contact constraints. The final solution is a set of joint angle changes for each frame in the simulation that should result in a realistic motion closely approximating the actual controller's movement, while taking collision constraints into account. The context of the implementation of these techniques is the BirthPlay application, which simulates uncommon or difficult birth operations to train obstetricians in the proper procedures. The implementation produces good solutions for basic inverse kinematics problems, but the collision response method results in extremely slow convergence for configurations with contacts.

2 Acknowledgements

I would like to thank the European Design Centre (EDC) in 's-Hertogenbosch for the opportunity to work on this project within the BirthPlay application. In particular, I am indebted to Gino van den Bergen, Steijn Kistemaker, and Peter van der Tak for their advice and support as my primary contacts at that company.

From Utrecht University, I would like to thank Arjan Egges, who was my personal supervisor during this project. I am also grateful to the supervisors and all participating students in our weekly discussions, who helped me many times with their advice and criticism.

This research owes a great deal to the work of Maaike Bartelink, who previously worked on BirthPlay at EDC and wrote her master thesis on the use of IK in that application [8]. Both her thesis and her work on the actual implementation were used as a basis for our own work and this report.

Table of Contents

3 Introduction

BirthPlay is a training program for obstetricians to practice difficult or uncommon situations that occur during birth. It is being developed by EDC (European Design Centre). The application teaches the use of proper procedures, including when and how to hold or grip the baby or use instruments. To make the simulation more realistic, the application uses two controllers that the user holds in his or her hands. The movement of the controllers is mirrored by the movement of the virtual doctor's hands. To make this possible, we use a mathematical method that takes the controller's position and orientation and calculates how the joints in the arm should rotate to match them. The solution should also take into account virtual constraints that do not exist in the real world (e.g. collisions). This is a difficult mathematical problem, and it is a challenge to find a solution that not only produces a stable and accurate result, but does so in real-time.

BirthPlay is an example of a 'serious game'. Though games are still mostly known as entertainment products, there is a growing number of developers creating games for purposes of training and education. The growing popularity of new input devices such as the Microsoft Kinect, Nintendo Wii Remote and Playstation Move, as well as similar devices for the PC, makes it possible to integrate the user's movement directly into game environments. This increases the number of tasks which can be simulated in serious games, and also makes them potentially more realistic and immersive. Figure 1 includes screenshots of the application, illustrating what it might look like in use.

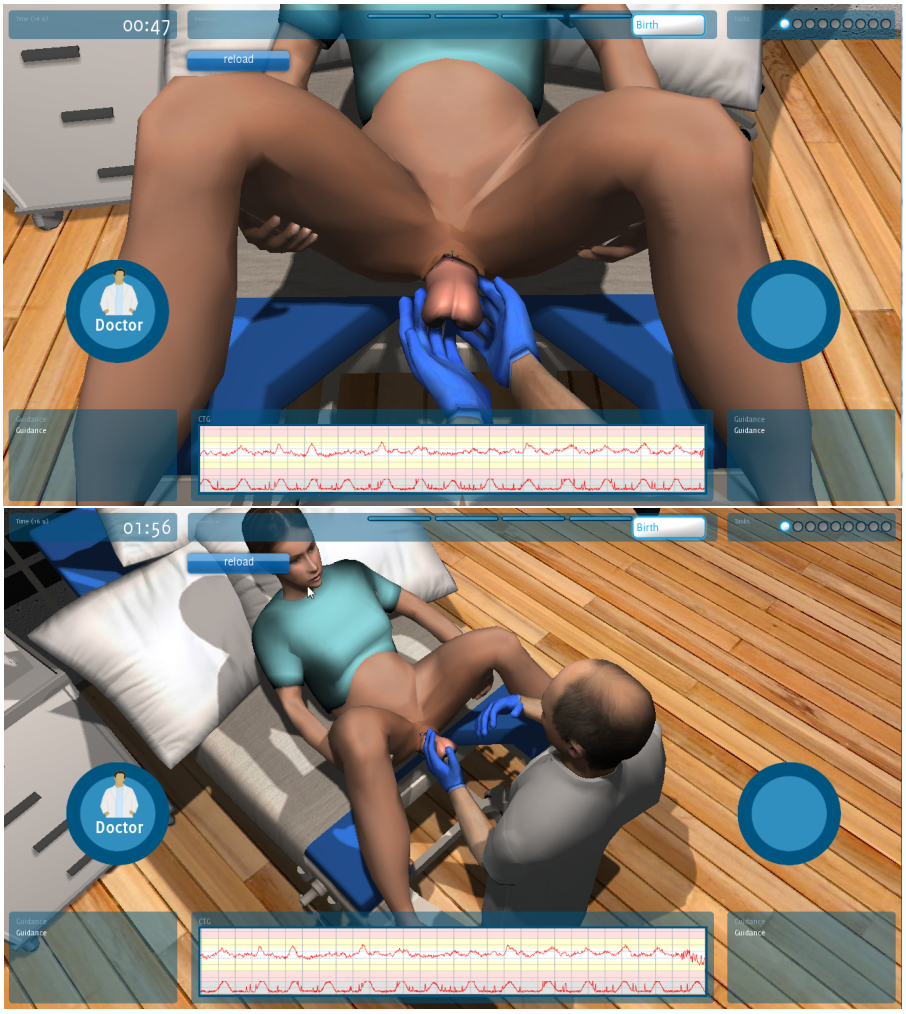

Figure 1: Screenshots of the (unfinished) BirthPlay application in first and third person. The movements of the doctor's hands reflect those of the user.

This research focuses on the Inverse Kinematics in the BirthPlay application. We want to improve the current methods to make the movement of the doctor more closely resemble the controller's position and orientation. We also want to improve the way in which collisions between the doctor's hands and arms and the mother and baby are handled, since the previous implementation has very unstable results for collisions. We want to make sure the solution in general is stable and there are no strange jumps or sudden movements in the simulation, since those greatly harm the simulation's realism. Of course, this all needs to work in real-time, with no (noticable) slowdown.

We will first discuss the basic concepts of data types and kinematics that are used in this project. After that, we summarize related work on inverse kinematics relevant to the project, in particular collision handling in the IK method. We then explain the exact IK methods used in the BirthPlay application, including the way collisions are handled. This also includes a representation of the algorithm in pseudocode. We also give a description of some particulars of the implementation, including the controllers, the skeleton model, the internal class structure and the external tools and libraries we employed. Finally, we discuss the implementation's results, the conclusions we can draw from them, and how parts of the method could be improved in future work.

3.1 Motivation

What we want to do in this project is to implement an IK method in the BirthPlay application that works in real-time. We want to improve on the current implementation on the key points of stability, joint limits, rotation solving, and most importantly, collision solving. Our aim in implementing this is to test existing IK methods in an actual application, where we need real-time solutions. The same goes for our implementation of collision solving within the IK method itself. We will go into the technical details of our implementation later, once we have established existing and discussed.

4 Background

This section will discuss some of the background theory that is necessary to understand the more specific techniques and implementations that are presented in subsequent sections. This discussion is divided into two parts. The data types section discusses the characteristics, advantages and disadvantages of some of the basic data types used to represent position and rotation data in 3D applications, some of which are also used in the BirthPlay application. The kinematics section discusses the basics of forward and inverse kinematics and the Jacobian and Cyclic Coordinate Descent methods for IK.

4.1 Data Types

This section discusses some of the basic data types which are used in 3D applications and simulations to store and manipulate position and rotation data. We give for each data type a formal definition and notation, and discuss its advantages and disadvantages. The data types that will be discussed are vectors for position and Euler angles, axis-angle, and quaternions for rotation. The details of this explanation are not strictly required to understand the methods discussed in this report, but are relevant to the actual implementation within the BirthPlay application. Those who are already familiar with these terms may prefer to skip most of this section and move on to Section 4.2.

4.1.1 Position: Vector

When working in three-dimensional Euclidean space, we naturally need at least 3 values to describe any position, one for each of the three dimensions. The 3D coordinate axes are usually denominated x, y, and z. They have the characteristic of being mutally orthogonal, meaning that they are all at right angles to each other. The simplest and most obvious way to define a certain position is as a combination *vector* consisting of three ordered scalar values that indicate the displacement along each axis, relative to the origin point. This is the representation used for all positions within the BirthPlay application. We define the three-value coordinate vector formally as follows:

$$\mathbf{v} = (\mathbf{x}, \mathbf{y}, \mathbf{z})$$

where $\mathbf{x}, \mathbf{y}, \mathbf{z} \in \mathbb{R}$.

4.1.2 Rotation: Euler Angles

The choice of representation for orientation and rotation is generally less obvious than the one for position, and often depends on the application. One point to note for all types of rotations is that they are not commutative in 3D space, meaning that two sequential rotations can have different results depending on the order in which they are applied.
A common way to represent a rotation in an 3D system is as a set of sequential rotations around the principal orthogonal axes (x, y, and z) and to represent the rotation as the set of rotation angles $(\theta_1, \theta_2, \theta_3)$, with each angle referring to one of these particular axes. This method is based on the fact that an arbitary rotation in 3D space has three degrees of freedom, so the three angles should be able to specify any rotation. The three angles are commonly called Euler angles. Using Euler angles, a rotation is accordingly defined as:

$$\mathbf{r} = (\theta_{\mathbf{1}}, \theta_{\mathbf{2}}, \theta_{\mathbf{3}})$$

where $\theta_{\mathbf{1}}, \theta_{\mathbf{2}}, \theta_{\mathbf{3}} \in \mathbb{R}$.

The advantage of this basic representation is that it is easily understandable at a glance and easy for the user to specify an orientation in most cases. The representation itself is efficient, since only 3 values are required. The operations needed to rotate a vector are simple compared to some other methods. Finally, there is also no normalization required for the angles, as is the case with some other methods. However, there is a major disadvantage in the fact that the use of only three degrees of freedom can lead to a coordinate singularity, usually called Gimbal lock. Two or more rotation axes can become aligned, causing certain desired rotations to become impossible until the lock is broken. Essentially, a degree of

freedom is temporarily lost. An illustration of this problem can be seen in Figure 2. The possibility of Gimbal lock in particular is the reason Euler angles are not commonly favored for use in 3D simulation.

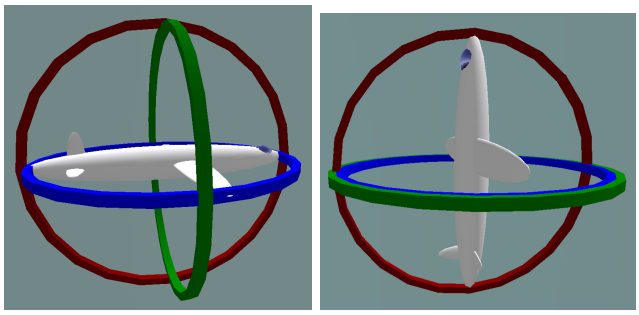

Figure 2: An illustration of Gimbal lock. When two axes align, one degree of freedom is lost. [4]

4.1.3 Rotation: Axis-angle

When using the axis-angle representation, an orientation is described as the pair of a 3-dimensional rotation axis and an angle, which represents the rotation around that same axis. This makes four elements in total. Storage can be made more efficient by multiplying the unit rotation axis vector with the value of the angle, making it only three. A formal definition of the axis-angle notation is as follows:

$$\mathbf{r} = (\mathbf{a}, \theta)$$

where $\mathbf{a}$ is a vector denoting the rotation axis and $\theta \in \mathbb{R}$ the rotation angle.

The main advantages of axis-angle representation are its efficiency and intuitiveness, since it directly represents the action of rotation. There is also no possibility of Gimbal lock. A disadvantage is that it is hard to compose a rotation using axis-angle representation without defining it using some other representation and converting it to axis-angle afterwards. Another issue is that there are an infinite number of valid angle choices for any one rotation. To prevent confusion, the angle is usually defined as lying in the interval $[-\pi, \pi]$, though this still leaves two possibilities. The use of this interval can cause issues when interpolating values, since at some point a 'jump' from $-\pi$ to $+\pi$. Furthermore, the identity rotation is not unique, since a rotation of 0 around any axis can represent it.

4.1.4 Rotation: Quaternions

A quaternion is a four-dimensional construct consisting of one real and three imaginary values. It is commonly written as:

$$\mathbf{Q} = \mathbf{w} + \mathbf{xi} + \mathbf{yj} + \mathbf{zk} \textit{ or } \mathbf{Q} = (\mathbf{w}, \mathbf{v})$$

where $v = xi + yj + zk$, $w, x, y, z \in \mathbb{R}$ and i, j, k are imaginary numbers such that:

$$\mathbf{i^2} = \mathbf{j^2} = \mathbf{k^2} = \mathbf{ijk} = \mathbf{-1}$$

The real component w is usually called the *scalar*, and the imaginary components xi, yj, zk the *vector*. A unit quaternion is a quaternion for which the square root of the product of the quaternion and its conjugation, or norm, is equal to one:

$$||\mathbf{q}|| = \sqrt{\mathbf{q}\mathbf{q}^*} = \sqrt{\mathbf{q}^*\mathbf{q}} = \sqrt{\mathbf{w^2} + \mathbf{x^2} + \mathbf{y^2} + \mathbf{z^2}} = 1$$

where $q*$ is the conjugation of the quaternion ($q^* = w - xi - yj - zk$). The quaternion conjugate is also used when calculating the quaternion combination of two separate rotations. The unit quaternion An axis-angle representation with angle θ and axis ω can be converted to a unit quaternion in the following way:

$$\mathbf{Q} = (\cos(\frac{\theta}{\mathbf{2}}), \omega \sin(\frac{\theta}{\mathbf{2}}))$$

and vice versa:

$$\theta = \mathbf{2} \arccos(\mathbf{w})$$

$$\omega = \frac{\mathbf{q}}{\sin(\theta/\mathbf{2})}$$

assuming $\theta \neq 0$, otherwise $\omega = 0$. These formulas also indicate by the way the sine and cosine of the angle θ are used when computing the quaternion that rotations modeled as unit quaternions have a natural unique representation, unlike those modeled using axis-angle representations, where θ has to be limited manually. We can think of the range of unit quaternion rotations as points on a 4-dimensional hypersphere (the 4 dimensions being w, x, y, and z). These formulas also indicate that rotations modeled as unit quaternions are unique. Using unit quaternions to represent rotations makes it easy to combine rotations, since the product of two unit quaternions is also a unit quaternion representing the combined rotation (though the order of the rotations does matter). Though quaternions are hard to compose by hand in all but the most simple cases, it is often easier to compose an axis-angle rotation and convert it to a quaternion than it is to compose a combination of Euler angles. Furthermore, since quaternion space is continuous, it is easier to interpolate smoothly between quaternions than it is with other representations. There is also no possibility of Gimbal lock. The main disadvantage of quaternions is that the raw values are basically incrompehensible to the user, since they contain imaginary numbers. This makes debugging more difficult.

4.2 Kinematics

This section discusses the basics of forward and inverse kinematics and the four different methods of inverse kinematics implemented in the BirthPlay application: Cyclic Coordinate Descent (CCD), Jacobian Pseudoinverse, Jacobian Transpose, and Damped Least Squares (DLS). Of these four, the latter three use the Jacobian matrix as a basis, and they are quite similar in underlying principles. We also give an explanation of the use of a spring constraint to ameliorate the pseudoinverse method's instability near singularities.

4.2.1 Basics: Forward Kinematics

The forward kinematics problem is: how to compute the next state of a given joint configuration given the current state and a set of joint parameters. These joint parameters can be any kind of change in the joints that affects the structure as a whole (in most cases these would be rotation and/or translation of the joints). Most joints in the human body would be modeled as having only rotational freedom, with translational freedom being more often found in robots.
To start with, we use the vector

$$\mathbf{\Phi} = [\phi_1, \phi_2, ..., \phi_M]$$

to represent the array of M joint degree of freedom (DOF) values. Furthermore, we use the vector

$$\mathbf{e} = [e_1, e_2, ..., e_N]$$

to represent the array of N DOFs that describe the end effector in world space. For example, a simple system solving for the position of the hand in a human arm chain, $\mathbf{e}$ would contain three DOFs (x, y, and z translations). More DOFs can be added to describe other characteristics of the end effector, such as its rotation. The *forward kinematic* function f computes the world space end effector DOFs from the joint DOFs, in other words, taking Φ as input and giving the values of $\mathbf{e}$ as output, and this is defined as follows:

$$\mathbf{e} = f(\mathbf{\Phi})$$

How f is itself defined depends on the system. For the simple human arm chain, it would be sequential rotation of each joint i by its DOF value ϕ_i. Each rotation influences later joints in the chain as well as the end effector (the final joint), and thus the results of subsequent rotations. The end result $\mathbf{e}$ is the end effector's final position. f can get much more complicated if the system has more types of joints and types of values in $\mathbf{e}$.

4.2.2 Basics: Inverse Kinematics

The inverse kinematics problem is basically the opposite of the forward kinematics problem; to take a given set of target end effector DOFs and figure out which change in joint DOF values will lead to that desired outcome. The problem can thus be defined as the mathematical inverse of the forward kinematics problem:

$$\mathbf{\Phi} = f^{-1}(\mathbf{e})$$

Finding and evaluating a good function f^{-1} is not a simple problem. There may be multiple possible solutions of $\mathbf{e}$ for a certain Φ, or there may be none at all. Certain methods may also have problems with different specific joint configurations. Because of this, there are many different methods of IK.

A major way of differentiating between methods of IK is whether they are *analytical* or *numerical*. Analytical methods simply invert the forward kinematics function to reach a solution. This method is exact, fast and guaranteed to be absolutely correct, but only viable for relatively simple chains. Numerical methods, on the other hand, converge on a solution by way of iteration and approximation. This approach is more costly, but it is generally applicable for most IK problems.

In most actual cases of IK use, one will find *under-constrained* IK problems, meaning that there are a large number of viable solutions that satisfy the constraints. For example, if the goal is a certain position/rotation of the hand, there will be multiple configurations of the arm and the rest of the body to satisfy that goal. At other times IK problems may be *over-constrained*, so that there are no viable solutions at all. An example is the case where the goal position is too far away, or only achievable if joints make impossible rotations. Both of these will ensure the analytical approach does not work, and we are forced to use numerical methods. The numerical methods implemented in the BirthPlay application are described in the next sections.

4.2.3 Cyclic Coordinate Descent

Cyclic Coordinate Descent is one of the easiest and most intuitive of commonly used inverse kinematics methods. The basic principle is to take one DOF at a time and optimize it to get as close as possible to the desired values of $\mathbf{e}$. This process is repeated for each DOF working up or down the joint chain until the solution is considered to be close enough to the goal. The basic working of CCD in a simple two-part 2D chain system is illustrated in Figure 3.

Since the 1-DOF problems are only concerned with a single operation, they can simply be solved by analytical methods. The CCD method has a number of advantages. It is easy to implement and easy to grasp. It is also stable around singular configurations and computationally cheap. However, it can easily lead to odd solutions, and the resulting motion may not necessarily be smooth, especially when working with complex systems.

4.2.4 Jacobian Methods

Since each of the remaining three methods is based on the concept of the Jacobian matrix, a brief description of its principles will be given below.
The Jacobian matrix is a vector derivative with respect to another vector. The basic principle is that it describes for each DOF its influence on the given target parameter(s), in the form of a partial derivative. Some examples of DOFs in IK are: a given joint's rotation (rotational DOFs), and its position (translational DOFs). The target parameter(s) are usually a specific position and rotation for certain joint or

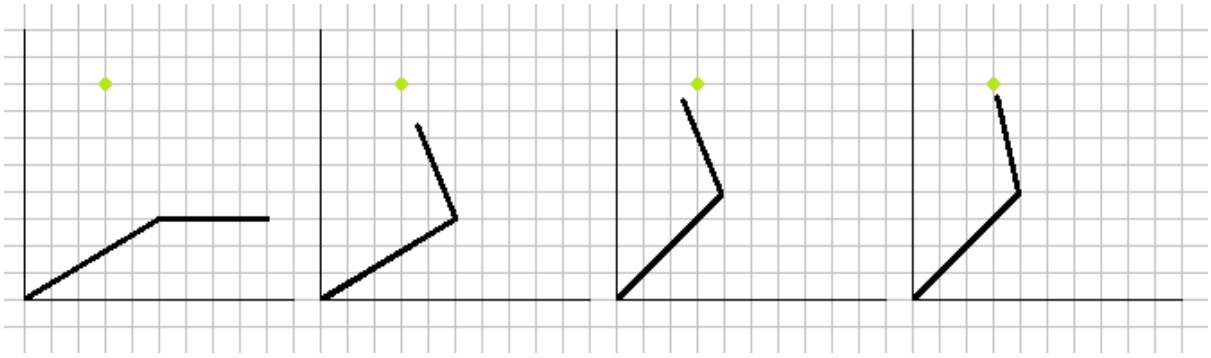

Figure 3: An illustration of a number of steps in the CCD algorithm for a simple two-joint 2D arm.

joints, but additional parameters can be added to further constrain the space of valid solutions. The Jacobian matrix can be written as:

$$J(\mathbf{f},\mathbf{x}) = \frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \cdots & \frac{\partial f_M}{\partial x_N} \end{bmatrix}$$

where $\frac{\partial f_i}{\partial x_j}$ is the partial derivative of f_i to x_j. The definition of the partial derivative is dependent on the system. An example of rotation and position derivatives in a joint chain is given in Section 6.1. Given a small change in joint DOFs, we can use the Jacobian matrix to compute the change in end effector DOFs:

$$\mathbf{\Delta e} \approx \mathbf{J} \cdot \mathbf{\Delta \Phi}$$

and, following from that equation:

$$\mathbf{\Delta \Phi} \approx \mathbf{J^{-1}} \cdot \mathbf{\Delta e}$$

The result is of course an approximation and is only valid if the change in end effector DOFs is small, since f is a nonlinear function. The partial derivatives in the Jacobian are based on the current joint chain configuration and will become invalid when the configuration changes drastically. Therefore, we want to take small steps towards the goal DOFs, recomputing the Jacobian with each iteration. An outline of the basic Jacobian IK technique therefore looks roughly like this:

```
while (e is too far from g) {
    Compute J(e, Φ) for the current pose Φ
    Compute J^-1
    Δe = β(g − e)
    ΔΦ = J^-1 · Δe
    Φ = Φ + ΔΦ
    Compute new e vector
}
```

where $\mathbf{g}$ represents the goal DOFs of the vector $\mathbf{e}$, and β is a variable between 0 and 1. β is used here to make the step ($\mathbf{g}$ - $\mathbf{e}$) toward the goal smaller, in order to prevent errors.

4.2.5 Jacobian Pseudoinverse

The basic Jacobian formula given above should theoretically give a valid solution for any small change in end effector DOFs. The only thing we need to compute the solution is the inverse of the Jacobian matrix. However, there is a major problem: computing the Jacobian Inverse is only possible if the matrix is square and its determinant is not equal to zero, which is often not the case. Therefore, a 'next-best' solution is to take the pseudoinverse instead. The pseudoinverse of a matrix is a generalized inverse that has the following properties (where A is a matrix and A^+ its pseudoinverse):

1. $\mathrm{AA^+A = A}$
2. $\mathrm{A^+AA^+ = A^+}$
3. $\mathrm{(A^+A)}^T = \mathrm{A^+A}$
4. $\mathrm{(AA^+)}^T = \mathrm{AA^+}$

expressed in text, this means that:

1. The product of the matrix and the pseudoinverse, then multiplied with the matrix, is equal to the matrix.
2. The product of the pseudoinverse and the matrix, then multiplied with the pseudoinverse, is equal to the pseudoinverse.
3. The transpose of the product of the pseudoinverse and the matrix is equal to the product of the pseudoinverse and the matrix. In other words, the product of the pseudoinverse and the matrix is symmetric.
4. The transpose of the product of the matrix and the pseudoinverse is equal to the product of the matrix and the pseudoinverse. In other words, the product of the matrix and the pseudoinverse is symmetric.

The pseudoinverse of a matrix also has the property of being unique. These properties describe the ways in which the pseudoinverse acts like the actual inverse in matrix multiplication. This behavior allows us to use the pseudoinverse as substitute for the actual inverse for the purpose of IK. The pseudoinverse of a matrix is defined in the general case as a 'nearest' solution to a system of linear equations lacking a unique solution. Applied to the Jacobian matrix $\mathbf{J}$, the pseudoinverse can be computed using the equation:

$$\mathbf{J^+} = (\mathbf{J^T J})^{-1}\mathbf{J^T}$$

or:

$$\mathbf{J^+} = \mathbf{J^T}(\mathbf{J J^T})^{-1}$$

This means we need to compute the inverse of $\mathbf{JJ^T}$, rather than the inverse of $\mathbf{J}$. The matrix $\mathbf{JJ^T}$ is guaranteed to be square. However, when at singularities the Jacobian matrix may not have a full row rank, because there is a direction of movement in the end effector that is not achievable. This means that the determinant of $\mathbf{JJ^T}$ becomes zero, which in turn means that the pseudoinverse cannot be computed. The algorithm as a whole is well-behaved in this case, and will not attempt to move in an impossible direction.
The pseudoinverse method might seem to be the best option for approximating J^{-1}, barring the actual inverse. However it has multiple problems. It is slow to compute, so it might not be a viable option if fast solutions are needed or when working with limited memory. It also suffers from instability in certain degenerate cases. Notably, though the algorithm is well-behaved *at* these singularities, it suffers from instability *near* them. These singularities occur if the derivative vectors line up and lose their linear independence (see Figure 4).

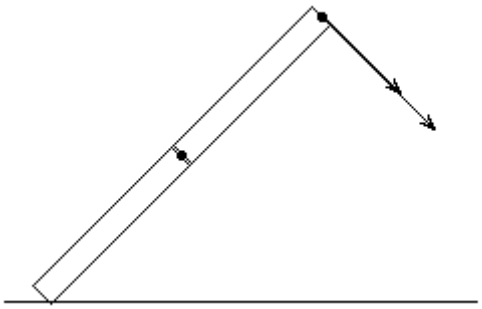

Figure 4: An illustration of the loss of DOFs near singularities.

4.2.6 Jacobian Transpose

The aforementioned problems with the pseudoinverse methods mean that, while it is probably the most accurate method for Jacobian IK, it is of limited practical use when working with real-time applications. The Jacobian Transpose method is a common alternative. The basic principle is simple: instead of taking the pseudoinverse of the Jacobian matrix, we simply take the transpose. So, we use

$$\mathbf{J}^{+} = \mathbf{J}^{\mathbf{T}}$$

instead of

$$\mathbf{J}^{+} = (\mathbf{J}^{\mathbf{T}}\mathbf{J})^{-1}\mathbf{J}^{\mathbf{T}}$$

Since the transpose is not a taxing operation, this gives a significant decrease in computation time compared to the inverse or pseudoinverse. An additional advantage of the Jacobian Transpose method is that we can loop through each DOF and compute its changes individually, instead of first looping through each DOF and computing the whole Jacobian matrix. This means that memory access and caching will also be reduced.
The method is also stable and works effectively with a single end effector. However, near singularities high joint velocities can result in irregular oscillations [20]. This typically occurs only when trying to fully extend the manipulator. It also converges slowly when using multiple end effectors [10].

4.2.7 Damped Least Squares

The final Jacobian method to be discussed is the Damped Least Squares method (DLS), which is also called the Levenberg-Marquardt method. The algorithm is similar to the Jacobian Transpose, with additional constraints. It works by minimizing the quantity

$$||\mathbf{J}\boldsymbol{\Delta\Phi} - \mathbf{e}||^2 + \lambda^2||\boldsymbol{\Delta\Phi}||^2$$

Where the damping constant $\lambda > 0$. This is achieved by setting

$$\boldsymbol{\Delta\Phi} = \mathbf{J}^{\mathbf{T}}(\mathbf{J}\mathbf{J}^{\mathbf{T}} + \lambda^2 I)^{-1}\mathbf{e}$$

Essentially, the algorithm 'leaves room' around the desired solution, with the strictness being dependent on both the damping constant and the changes in joint DOFs. The value of the damping constant depends on the details of the joint configuration and target positions. It should be large enough so that the solutions of the algorithm are stable near singularities. However, if it is too large, the convergence rate will be too slow. The performance of DLS should be similar to the Jacobian Pseudoinverse when away from singularities, and avoid the pseudoinverse's problems near singularities [9]. Compared to the Jacobian Transpose, it should be slower in computation time, but provide accurate results in less iterations since its approximation of the Jacobian Inverse is better (like the pseudoinverse method itself).

4.2.8 Spring Constraint for the Pseudoinverse Method

As mentioned in the section on the Jacobian pseudoinverse method, the pseudoinverse method suffers from instability near singularities. One way to ameliorate this problem is to use a soft spring constraint. This approach is similar to DLS, but with a different damping method. It is based on physics methods used to soften constraints between rigid bodies (used in the Box2D physics engine among others ([11])). The softening prevents springs in a physics system from 'blowing up', creating large and unrealistic reactive forces. The same principle can be applied to improve the pseudoinverse method's stability.

We start with the differential equation of a damped harmonic oscillator. The harmonic oscillator is a system which, when displaced, experiences a proportional force countering that displacement. The damped harmonic oscillator adds damping, so that the force and displacement are gradually reduced (as is the case with most real-life oscillators). The differential equation is as follows:

$$\mathbf{m}\frac{\mathbf{d^2x}}{\mathbf{dt^2}} + \mathbf{c}\frac{\mathbf{dx}}{\mathbf{dt}} + \mathbf{kx} = \mathbf{0}$$

Where m equals mass, c the damping coefficient and k the spring stiffness. The equation is an expression of Newton's law, where force equals mass times acceleration. The three terms of the equation express acceleration, velocity and position. Now, we can simplify the equation by introducing the damping ratio (ζ, which controls the amount of oscillation in the solution) and angular frequency (ω, which controls the rate of oscillation), reducing the number of constants to two:

$$\frac{\mathbf{d^2x}}{\mathbf{dt^2}} + \mathbf{2}\zeta\omega\frac{\mathbf{dx}}{\mathbf{dt}} + \omega^{\mathbf{2}}\mathbf{x} = \mathbf{0}$$

$$\mathbf{2}\zeta\omega = \frac{\mathbf{c}}{\mathbf{m}}, \omega^{\mathbf{2}} = \frac{\mathbf{k}}{\mathbf{m}}$$

A small ζ will allow unhindered oscillation, while a large value causes the oscillation to go to zero over time. If $\zeta = 1$, all oscillation will be gone, with larger values only slowing down the mass. Note also that both ζ and ω are dependent on the value of m, the mass. We can solve the harmonic oscillator using a semi-implicit Euler integrator to make the solution reliably stable, and obtain the following formulas that relate the softness parameters to the spring and damper constants.
The value gamma softens the velocity constraint:

$$\gamma = \frac{\mathbf{1}}{\mathbf{c} + \mathbf{hk}}$$

Similarly, the value beta controls position error (keeping the position centered on the origin) and stores energy in the 'spring', and is defined as:

$$\beta = \frac{\mathbf{hk}}{\mathbf{c} + \mathbf{hk}}$$

In the above formulas, $\mathbf{k}$ is constant and set to some predetermined value evaluated to give the best results (actually computed using effective mass and frequency, but effective mass is assumed to be equal to 1 in the BirthPlay application, since it is not a physics simulation). The timestep $\mathbf{h}$ is dependent on the simulation and its update frequency. The factor $\mathbf{c}$ is dependent on damping constant ζ and the frequency ω, like so:

$$\mathbf{c} = \mathbf{2}\zeta\omega$$

We set the value of $\zeta = \mathbf{1}$, for optimal convergence (critical damping). If a different damping constant was chosen the constraint would act like a conventional spring, 'bouncing' around the central point. Since $\mathbf{k} = \omega^{\mathbf{2}}$, we get:

$$\mathbf{c} = \mathbf{2}\sqrt{\mathbf{k}}$$

And so all variables are expressed as a formula dependent on the value of k. The value of k itself must be chosen by the programmer, but since the influence of time step and mass has been taken into account, there should be no need to alter it after changes to the program, even if environment parameters change. In the Jacobian IK method, the value β is applied to the target vector, while γ is applied to the Jacobian.

$$\mathbf{J} = \mathbf{J} + \mathbf{I}\gamma$$

Where I is a matrix with the same number of columns and rows as the Jacobian matrix, in which the only nonzero values are the diagonal ones, which have a value of 1. The Jacobian is analogous to a rigid constraint in the spring model. Using the extra factor of γ to compute the new Jacobian makes the position constraint less rigid, so that it shouldn't 'blow up' for configurations with singularities. It has a similar effect to the that of the λ factor introduced by the DLS method. The target vector is analogous to the displacement velocity in the harmonic oscillator. Multiplying the vector by the value of β compensates for the position error introduced by γ. After the alteration to the Jacobian, the inverse is calculated in the normal way.

4.3 Summary

We discussed some of the basic data types used in 3D simulation and their advantages and disadvantages. Vectors for position and Euler angles, Axis-angle and Quaternions for rotation. We also reviewed the basics of inverse kinematics and gave a more in-depth explanantion of the Cyclic Coordinate Descent method (CCD) and the Jacobian methods (Jacobian Pseudoinverse, Jacobian Transpose, and Damped Least Squares (DLS)). In the next section we will go into previous related work in IK and collision IK in particular. We will also this project's contribution to IK research.

5 Related Work

In this section we will discuss some of the research that has been previously done on Jacobian inverse kinematics in general and collision handling in particular. We also give a motivation for our research, defining the main goals of the project.

Firstly, there is the previous work related to basic inverse kinematics. The basic IK problem is to find a configuration of the various degrees of freedom in a system that satisfies a set of constraints. A practical example is configuring the rotation of a number of joints in a model of the human arm (degrees of freedom) to have the model's hand reach a certain position and orientation (constraints). There are many methods of IK, which handle the problem in different ways. Some of them support additional constraints besides position and orientation of the end effector, like collision constraints. Collision constraints are used to prevent configurations where there are collisions between the body being manipulated and the environment. Naturally, finding a solution that satisfies these constraints may require additional systems and computation time.

Of the different methods for solving IK problems, many come originally from robotics applications. This report focuses on Jacobian IK methods, in particular the Jacobian pseudoinverse method, the Jacobian transpose method, and the Damped-Least-Squares (DLS) method, also called the Levenberg-Marquardt method [18]. These methods were explained in section 4, and a good summary can also be found in [9]. The basic CCD [19] (Cyclic Coordinate Descent) method for IK was also implemented in the BirthPlay application, but is mainly used for purposes of testing and comparison. A more detailed description of these techniques is given in Section 4.2, which discusses kinematics.

The Jacobian IK techniques we use require the computation of the Jacobian matrix, or an approximation thereof. We can do this in real-time, since the complete chains of joints we simulate for consist only of one limb at most. However, doing this kind of computation for the whole body, or multiple bodies, would be cost-prohibitive in real-time. For simulations that require the simulation of a great number of complex systems in real-time, more simplified approaches are needed. An example of such an approach is given by Kulpa et al. [12]. This approach uses a morphology-independent motion representation that simplifies the skeleton to only a few key subparts with associated constraints (see Figure 5). In the BirthPlay application, the key constraints would be the fixed position of the shoulder and the position and orientation of the wrist joint. The representation also contains constraints that are characteristic of the motion, such as foot contact with the ground. Coupled with the use of the CCD algorithm to quickly iterate to new solutions while manipulating body segments instead of the kinematic chain, this allows for the animation of multiple bodies with different morphologies in real-time with varying constraints.

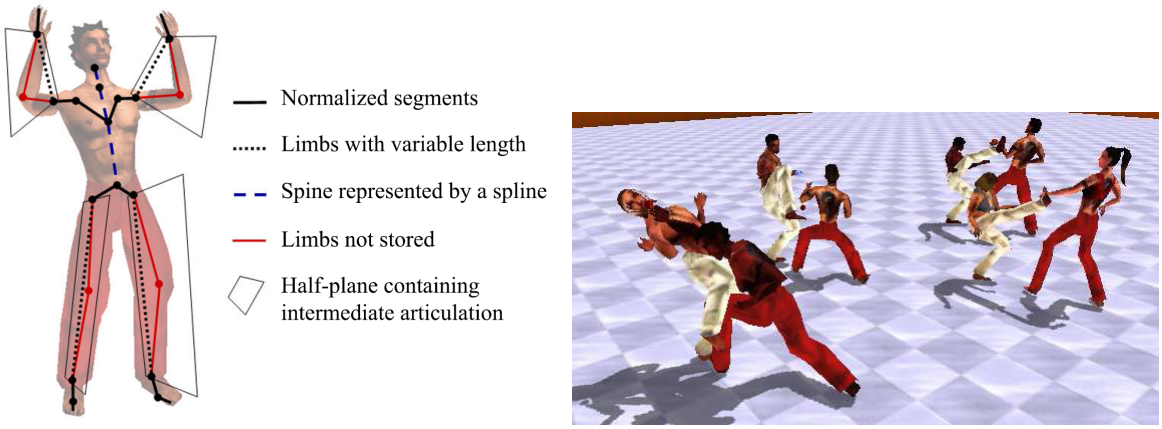

Figure 5: An illustration of the morphology-independent normalized skeleton representation and simulation of multiple characters animated in real-time with varying constraints based on the opponents' positions. [12]

Another technique is the use of Priority-Based IK to balance simulation constraints. There are many

different kinds of constraints, based on the nature of the simulation or simulated body. Examples are feet position constraints to prevent sliding during standing animations, balance constraints, constraints to restrict joint rotation, and constraints that are specific to the animation, such as which way the eyes are facing. When there is no possible solution that satisfies every possible constraint, some constraints have to be more important than others. For example, if the choice is between reaching a certain target configuration and violating a collision constraint, the collision constraint will normally be more important. A common way of handling priority of contraints is to use a weight strategy. The most basic approach to Jacobian IK has all constraints using the same weights, with the solution being a compromise between them with the least possible total violation of all constraints. Using the weighted method, different weights are assigned to each constraint, and the solution is a compromise between them based on those weights. The greater the difference in weights, the more the solution will tend towards satisfying certain constraints. Examples can be found in [21],[15],[6].
This method has the advantage of being easily implemented, with the only operation necessary being to multiply the target vector's values by the pre-determined weights. There is no need for additional computations or alterations to the Jacobian itself. Weights may also be determined differently on each frame if necessary, to alter the system's behavior depending on the current configuration. However, it has a great drawback in that it cannot guarantee that any constraint is satisfied (if at all possible) unless all other constraints are completely eliminated from consideration by reducing their respective weights to zero.

Another way of handling constraint priorities is to use multiple priority levels. Using this method, every constraint is satisfied as much as possible, but ultimate priotity is given to constraints with a higher priority level. An thorough explanation and example implementation is given in [7]. This method can guarantee the satisfaction of high-level constraints without a compromise solution. This is done through the construction of a projection operator onto the Null space of the Jacobian (as proposed by Liégeois [13]), which allows for an additional constraint to the solution based on the higher priority. An illustration of the difference between the methods is given in Figure 6, and an illustration of the multiple priority levels method in action in Figure 7. A drawback of using multiple priority levels is higher computational cost and complexity of the resolution methods. It is also less reliable, since convergence to an optimal solution is not always guaranteed.

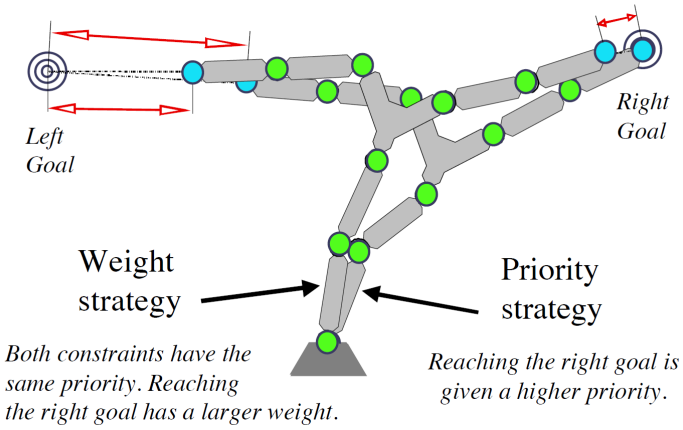

Figure 6: An illustration of the difference between the weighted and priority-based approaches, using a chain with two end effectors.[7]

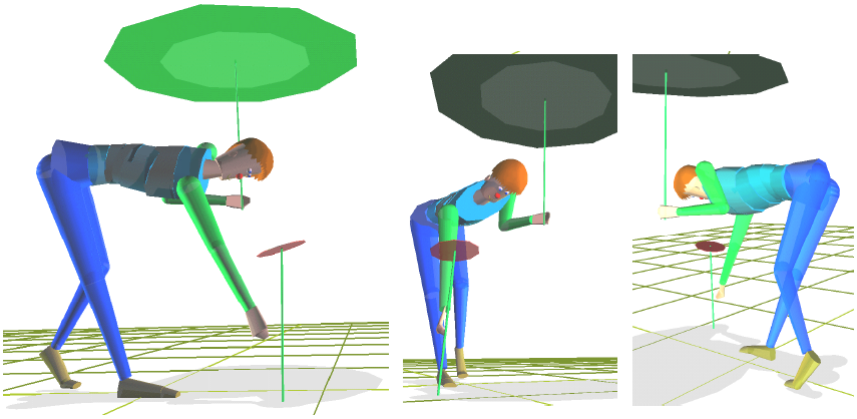

Figure 7: An illustration of the multiple priority levels method, combining balance, reach, and gaze constraints while holding the umbrella vertically (with four priority levels).[7]

When discussing collisions, we generally make a division between collision detection and collision handling. Given a configuration of the body being manipulated and the environment and/or other bodies, there are multiple ways to detect collisions, from rudimentary approaches using intersection tests between simple boxes or spheres to more advanced algorithms such as the GJK (Gilbert-Johnson-Keerthi) algorithm, which can compute the distance between any two convex objects [17]. Space Partitioning, using cubes, voxels, grids, and tree structures, is also used to reduce the number of collision tests needed.

Once the collision occurs and is detected an effort must be made to prevent it from occurring. This can be done either by working from the current configuration, which contains a collision, or by reverting to a previous, collision-free configuration. Either way, the existing constraints must be altered to prevent the collision, or a new constraint must be added to the simulation. That new constraint must avert the collision while compromising the accuracy of the simulation as little as possible. In simulations that use physics, this constraint may take the form of an actual force, as described by Newton's second law, while other simulations may need to 'cheat' in various ways to approximate a constraint that results in a natural solution.

The use of Jacobian IK methods for collision solving is of particular interest to our research, since that is the main improvement needed in the BirthPlay IK system to reach a more realistic simulation. Since we are working in a real-time simulation, we prefer local optimization methods as opposed to global optimization methods, since the latter are much more expensive. [21] gives an example of the use of an optimization method for IK (though not Jacobian IK). The degree of satisfaction of all constraints is described as a nonlinear function, which is then minimized under another set of constraints describing the joint limits. An application of this method is implemented in the *Jack* system [15].

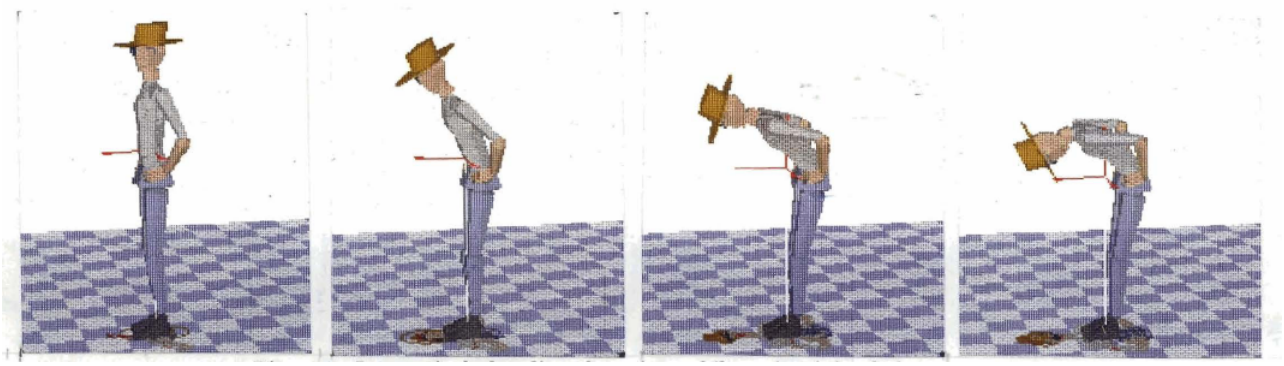

Figure 8: An illustration of the Jack system in action, bending the spine while maintaining the balance constraint. The same method could be applied to maintain a contact constraint while moving. [15]

When collisions are prevented using constraints, the priority of those constraints is normally higher than that of the other constraints. It is rarely acceptable for a simulation to contain a visible and obvious collision between one body and another body or a part of the environment. When using the weight-based method for priority, this means that the collision constraints are given a very high weight value relative to other constraints, and those constraints may even be reduced to zero until the collision is no longer present. When using multiple priority levels, the collision constraint can simply be prioritized over most others.

Yet another way to handle collisions is to use obstacle avoidance, so as to prevent collisions from happening in the first place. One approach to this proposed in [14] is based on using redundant manipulators, intended for use in robotics, such as in construction. Redundant manipulators have more than the normal six degrees of freedom (for position and rotation). The extra DOFs result in greater flexibility and dexterity in the motion specification of the manipulator. This approach does require that at each step the closest point to the obstacle is found, and a velocity vector is computed directly away from the obstacle surface. Satisfying this velocity vector is treated as a secondary goal to the end effector's velocity goal. Once the primary goal is satisfied, the system's redundancy is used to avoid obstacles as much as possible. The constant extra computation is computationally more expensive and difficult than only resolving collisions, but leads to more realistic motion. Additionally, it can of course not be guaranteed that collisions will be completely prevented, so we do also need an actual collision handling method to supplement this approach. An illustration of this method is given in Figure 9.

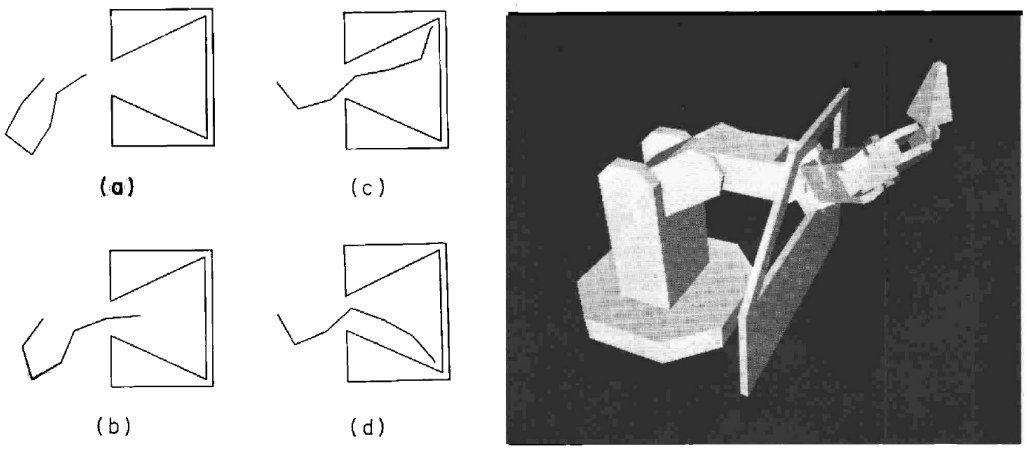

Figure 9: Left: An illustration of the method using a redundant manipulator for obstacle avoidance, working in a cavity while avoiding contact. **Right:** A 3D redundant manipulator operating through the window of an automobile door. [14]

5.1 Contribution

This project's contribution to IK research is to reach a solution for collision-aware inverse kinematics that can be used in the real-time BirthPlay application. The previous implementation of IK in the BirthPlay application already provides IK solutions using the CCD method and Jacobian IK. However, it still has multiple issues in terms of capability and performance that we need to address in our own implementation.
Firstly, the IK method implemented in the BirthPlay application previous to our research does not support rotation derivatives in the Jacobian matrix, instead solving for rotation only on the last joint in the chain. This being the case, our first step is to find and apply a way to also take into account the rotation of other joints in the chain. This much we can do easily relying on existing methods (these methods were explained in detail in Section 4.2 of this report).

A second issue with the current IK solution in the BirthPlay application is the way joint angle changes are limited to a certain natural range. This is done at the level of joint rotation itself. In this way, it can be guaranteed that there is no rotation that exceeds the prescribed bounds. However, since the limits are not taken into account at the IK level, the solution provided by the IK algorithm may be suboptimal given the actually available range of motions. This problem does not result in wildly unrealistic motions in the previous solution, but we can nonetheless feel instinctively that the algorithm might be improved if the limits were actually taken into account. Our aim is to try to find a way to do this in the new algorithm.

Finally, there is the issue of collision handling. The previously implemented IK method in the BirthPlay application does provide a solution to this issue, but not one that satisfactorily works in real-time simulation, nor generates a believable and realistic motion. Our first and most basic aim will be to include the collision parameters in the Jacobian to avoid collisions. We will then implement the weighted approach to prioritize collision constraints and examine the results. Due to time constraints the approach of multiple priority levels is not implemented, though it may give good results.

5.2 Summary

In this section we discussed some of the previous research on IK in general and collision IK in particular. We saw examples of the application of IK in simulations and robotics. We saw multiple examples of collision handling in IK, using obstacle avoidance and priority methods based on weights and discrete priority levels. Finally, we explained our this research's aims: to improve the current IK solution in BirthPlay at several points and provide a realistic, real-time solution to collisions. In the next section

we will explain exactly how the Jacobian matrix is constructed and used in the BirthPlay application and how collisions are avoided using the Jacobian to achieve a more natural motion.

6 Collision-Aware Inverse Kinematics

This section discusses the Jacobian solving algorithm that was developed and implemented for the BirthPlay application. It examines how the DOF derivatives that make up the Jacobian matrix are computed and how collision handling is done within the Jacobian. On the whole we follow the description of Jacobian IK outlined in Section 4.2.4, but there are particulars in the implementation and the BirthPlay system that force us to make choices regarding the modeling of certain variables and values. Pseudocode representations of the algorithms used in the BirthPlay application are included, as well as examples of the Jacobian matrix with the location of each derivative to better illustrate the resulting matrices. Finally, we explain some additional techniques that were implemented to help with particular issues of the existing methods.

6.1 Jacobian Construction

The first issue is the construction of the Jacobian itself. The basic form of the Jacobian, as explained earlier in Section 4.2.4, is this:

$$J(\mathbf{f},\mathbf{x}) = \frac{d\mathbf{f}}{d\mathbf{x}} = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_N} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \cdots \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\partial f_M}{\partial x_1} & \cdots & \cdots & \frac{\partial f_M}{\partial x_N} \end{bmatrix}$$

We know that each of the N degrees of freedom (DOF) influences the target parameter, and the Jacobian matrix contains the influence for that DOF. We need to first decide exactly what kind of values the target parameters and DOFs will be, and how we will represent them. Luckily this is easily done given the aim of the simulation. We know that the system will be used to visualise the user's hands' positions and rotations. Therefore we can take as the target parameters the position and rotation of the wrist joints (to represent the hand as a whole). We will refer to this joint as the end effector. The position of the end effector is represented as a 3-dimensional vector with x, y, and z values, and the rotation as an axis-angle representation, with x, y, and z values (the angle value of the rotation is stored as the magnitude of the axis vector). We therefore have a total of 6 target parameters for the end effector in this implementation (M = 6).

We now know the target parameters, which makes finding the DOF values also easy. Since the target parameters are the end effector's position and rotation, we simply take as Jacobian DOFs the rotations of each joint in the arm chain that influences that position and that orientation. We also make a choice to limit the simulated doctor's motion to only the arms, in order to keep it simple, reliable, and efficient in computation time. Thus each hand moves independently of the other, and we can use different Jacobian matrices for each arm chain.

Finally, we must choose how we model the influence itself. We know that the rotation (change in DOFs) of the joints is what causes changes in position and rotation in the hand. We could choose to use this rotation (in degrees or radians) as the DOF value. However, we choose in this implementation to use the angular *velocity* instead, so that the rotation is relative to the frame rate of the simulation. This means we take the rotation over time instead of the rotation itself. The effective benefit of this is that the simulation will not progress faster or slower depending on whether the program has more or less computation time available (within reasonable limits, of course). The resulting motion should therefore be more natural, without sudden changes in movement speed. In practice, this simply means that the rotation is calculated as normal, but the target velocity is divided by the time step value. It makes no difference in the construction of the Jacobian itself.

Figure 10 depicts a small joint chain. We define the following variables:

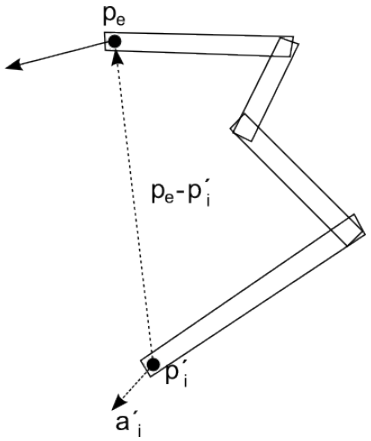

Figure 10: A small joint chain with associated variables.

1. a_i': unit length rotation axis for joint i in world space
2. p_i': position of joint i in world space
3. p_e: end effector position in world space

We define the rotational DOF's influence on the end effector's position as the change in the position of e when rotating around axis a_i'. The vector $p_e - p_i'$ is the distance vector or rotation 'arm' of the rotating joint and the end effector. Therefore, we can use the cross product to compute the rotation derivative:

$$\frac{\partial \phi_\mathbf{i}}{\partial \mathbf{p_e}} = \mathbf{a_i'} \times (\mathbf{p_e} - \mathbf{p_i'})$$

Where ϕ_i is the DOF value. For the end effector's rotation, we can simply use the axis a_i' itself as the derivative, since the change in rotation is not dependent on the end effector's position. Note also that the formula indicates that the joints in between the rotating joint and the end effector have no influence on the result.

6.1.1 Algorithm for Jacobian Construction

A description of the algorithm used to construct the basic Jacobian matrix in the BirthPlay application is given below. We first define the following method for easier comprehension. Its function is simply to add the values of two given vectors with three values each to the Jacobian (one for position and one for rotation), constructing one column that represents the influence of one degree of freedom on the end effector.

Algorithm 1 Adding a column to the Jacobian

```
1: function AddColumn(Jacobian J, Vector3 v_p, Vector3 v_r, int index)
2:     J[0, index] = v_p.x;
3:     J[1, index] = v_p.y;
4:     J[2, index] = v_p.z;
5:     J[3, index] = v_r.x;
6:     J[4, index] = v_r.y;
7:     J[5, index] = v_r.z;
```

This method is used in the following, the main algorithm. It constructs the whole Jacobian using the formulas explained earlier in this section. The DOF values for each joint are calculated in turn and added to the matrix. The DOFs' influence on the end effector is of course only computed and added to the Jacobian if the joint actually has that degree of freedom.

Algorithm 2 Constructing the Jacobian matrix

```
1:  float[ , ] J;
2:  int colIndex = 0;
3:  for i = 0..n do
4:      //compute derivatives:
5:      if Joint i has X DOF then
6:          dPx = a'_i x × (e − p'_i);
7:          dRx = a'_i x;
8:      if Joint i has Y DOF then
9:          dPy = a'_i y × (e − p'_i);
10:         dRy = a'_i y;
11:     if Joint i has Z DOF then
12:         dPz = a'_i z × (e − p'_i);
13:         dRz = a'_i z;
14:     int colsAdded = 0;
15:     //add derivatives to Jacobian:
16:     if Joint i has X DOF then
17:         AddColumn(J, dPx, dRx, colIndex);
18:         colsAdded = colsAdded + 1;
19:     if Joint i has Y DOF then
20:         AddColumn(J, dPy, dRy, colIndex + colsAdded);
21:         colsAdded = colsAdded + 1;
22:     if Joint i has X DOF then
23:         AddColumn(J, dPz, dRz, colIndex + colsAdded);
24:         colsAdded = colsAdded + 1;
25:     colIndex = colIndex + colsAdded;
```

In the above algorithm n is the number of joints in the chain. $a'_i x$ is the local x-axis for joint i ($a'_i y$ and $a'_i z$ are the local x- and z-axes, respectively). See also Figure 10 and the accompanying explanation if the computations in this algorithm are unclear. The computed values stored in the derivative vectors are themselves vectors, each with x, y, and z values. For the BirthPlay application, the joints in each individual arm chain are the two joints in the shoulder, the elbow joint, and the wrist joint (four joints in total). All joints are modeled as having three DOFs, except for the elbow, which is a hinge type joint. For this configuration, the resulting Jacobian when using Algorithm 2 would look like this:

$$\left[\begin{array}{ccc|ccc|c|ccc} l_1x, p_ex & l_1y, p_ex & l_1z, p_ex & l_2x, p_ex & l_2y, p_ex & l_2z, p_ex & l_3z, p_ex & l_4x, p_ex & l_4y, p_ex & l_4z, p_ex \\ l_1x, p_ey & l_1y, p_ey & l_1z, p_ey & l_2x, p_ey & l_2y, p_ey & l_2z, p_ey & l_3z, p_ey & l_4x, p_ey & l_4y, p_ey & l_4z, p_ey \\ l_1x, p_ez & l_1y, p_ez & l_1z, p_ez & l_2x, p_ez & l_2y, p_ez & l_2z, p_ez & l_3z, p_ez & l_4x, p_ez & l_4y, p_ez & l_4z, p_ez \\ \hline l_1x, r_ex & l_1y, r_ex & l_1z, r_ex & l_2x, r_ex & l_2y, r_ex & l_2z, r_ex & l_3z, r_ex & l_4x, r_ex & l_4y, r_ex & l_4z, r_ex \\ l_1x, r_ey & l_1y, r_ey & l_1z, r_ey & l_2x, r_ey & l_2y, r_ey & l_2z, r_ey & l_3z, r_ey & l_4x, r_ey & l_4y, r_ey & l_4z, r_ey \\ l_1x, r_ez & l_1y, r_ez & l_1z, r_ez & l_2x, r_ez & l_2y, r_ez & l_2z, r_ez & l_3z, r_ez & l_4x, r_ez & l_4y, r_ez & l_4z, r_ez \end{array}\right]$$

For links l_1 and l_2 (shoulder), l_3 (elbow) and l_4 (wrist). It can be seen that the each column corresponds to one DOF in the joint chain, while each row corresponds to one DOF of the end effector. This is the basic form of the Jacobian matrix.

6.2 Collision handling in the Jacobian

This section discusses the way in which collisions are detected and resolved in the BirthPlay application. We speak of a collision when two bodies in the simulation intersect. There are a number of different entities with bodies in the BirthPlay application. The important ones when concerned with collisions are the obstetrician, the mother, and the baby. Collisions will most frequently involve the obstetrician,

and in particular their arms and hands, since they follow the user's movements directly. Other objects are mostly static in the current implementation, and those that do move do so using pre-composed animations, not our IK method.

6.2.1 Collision Shapes

There are two basic options: using the models themselves as collision shapes, or using collision 'masks' that approximate the actual shape of the model, but have a simpler structure. The former is absolutely correct for the given shapes, but can be inefficient when the models have high fidelity. Since the BirthPlay application works in real-time, this option is not attractive for our purposes. The latter method will keep computation times low and have little impact on the believability of the simulation, provided that the collision shapes are chosen well.

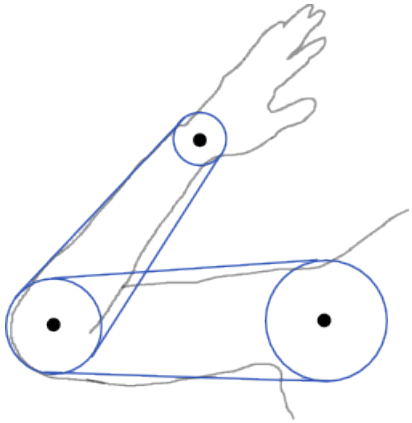

Figure 11: An illustration of collision shapes in the BirthPlay application.

When using collision masks, the most frequently used shapes are simple geometric bodies, like boxes and spheres. For the BirthPlay app, the collision shapes are based on spheres for each joint, with connecting cylindrical shapes between the joints. Figure 11 illustrates the resulting collision shape. This capsule shape is well-suited to approximate the shape of limbs. Since the base is a bounding sphere rather than a bounding box, there will never be a large change in contact normal direction when there is only a small change in contact position, which improves the stability of the IK solutions. It is also convex in all places, making it less likely that different collision shapes will get caught in each other. This is important, since there is no real path planning and the only way to resolve situations where shapes become stuck would be through the user's influence.

6.2.2 Collision Detection

In the BirthPlay application, there is a check for collisions in between every frame of the simulation. If there is an intersection between two different collision shapes, this is stored as a contact. This contact contains information relevant to the collision that may be used during the collision response. The variables that define the contact are:

1. p_c: the contact's position
2. N: the contact's normal vector (unit length)
3. d: the contact's collision depth

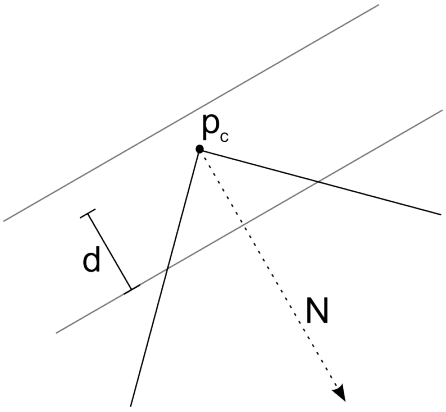

Figure 12: An illustration of contact variables in context. The angular shape is the one colliding.

Figure 12 presents the contact variables in context. Note that there is only one contact per collision between two unique shapes. In other words, two shapes cannot collide twice in the same frame. This is not a problem in our simulation, since the nature of the standard collision shape prevents situations where multiple contacts would be needed.

6.2.3 Collision Response

Once the collision has been detected, a response from the system is necessary to correct it in further iterations of the simulation. The aim is to prevent collisions entirely to ensure realism. To prevent the contact, we need to steer the colliding shape away from the contact point. As was explained, we know the contact point's location, its direction or normal vector, and the contact depth or magnitude. Using these variables, we can construct an impulse velocity, which we add to the Jacobian matrix itself. The impulse velocity is a vector with the same direction as the contact normal and a length equal to its depth.

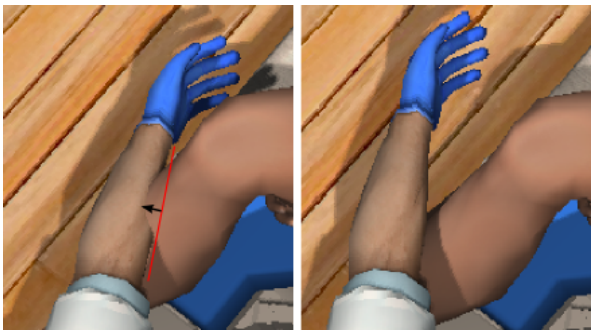

Figure 13: An illustration of contact resolution in the BirthPlay application. The contact effector's target velocity is indicated in the first picture.

We effectively add an additional effector to the Jacobian matrix. It can be thought of in the same way as the end effector we normally solve for. Its location is the location of the contact, and its target velocity is a vector in the direction of the normal, with a magnitude equal to the contact depth. If this target is reached, the new location of the contact will no longer be within the collision shape, so there will be no more contact in the next iteration. This method can be used for any number of contacts. Each simply gets its own effector value in the Jacobian matrix and target velocity. Since we solve for contact position, this means we add 3 rows to the matrix per contact (for the x, y, and z values of the position vector). Because the Jacobian matrix is reconstructed at each time step, the contact effectors will also be constructed anew if necessary, but will not remain for more than one step.

The Jacobian matrix now has multiple effectors and multiple target inputs. The optimal solution is of course one that satisfies all constraints for all end effectors, but such a solution may not be possible in

many cases, especially if there are multiple contacts. If there is a conflict between different targets, the jacobian solver will try to find the closest possible solution. This means that the *total* difference between the target positions/rotations and the actual positions/rotations of the end effectors is minimized. The solution will necessarily strike a balance between solving for each of the different effectors. A problem which then occurs is that solving for the end effector's position and rotation and solving for no contacts can result in a balanced solution that is not contact-free.

The method used to prevent this in the BirthPlay application is to iteratively reduce the magnitude of the end effector's position/rotation velocity as long as contacts are present in the new configuration. Eventually, the end effector velocity is so far reduced that the contact impulse 'wins out'. This is the 'weighted' approach also mentioned in Section 5. However, this results in slow convergence when such conflicts do occur. Furthermore, there may be configurations with multiple contacts where a solution with no contacts at all is not possible (especially if more than one collision shape can move). Therefore, the number of iterations must also have an upper limit. The exact value of this limit will depend on the amount of time per frame afforded to the IK method.

6.2.4 Algorithm for Jacobian Collision Handling

The following is an updated algorithm that adds the contact derivatives to the Jacobian matrix construction. Its function is mostly identical to that of the algorithm explained in Section 6.1.1. The values of the DOFs' influence on the end effector are added in the same way and in the same place. The additional operations add the DOFs' influence on the contact effectors to the Jacobian. As can be seen, the operation uses the same formulas to compute the derivative values, only replacing the end effector's position with that of the contact effector. Note that the row index is raised by three with the addition of each new contact, so that each new set is added below the last. In this way, the resulting Jacobian will have the same amount of columns, but three extra rows per contact.

Algorithm 3 Constructing the Jacobian matrix (with contacts)

```
1: float[ , ] J;
2: int colIndex = 0;
3: for i = 0..n do
4:     //compute derivatives:
5:     if Joint i has X DOF then
6:         dPx = a'_i x × (p_e − p'_i);
7:         dRx = a'_i x;
8:     if Joint i has Y DOF then
9:         dPy = a'_i y × (p_e − p'_i);
10:        dRy = a'_i y;
11:    if Joint i has Z DOF then
12:        dPz = a'_i z × (p_e − p'_i);
13:        dRz = a'_i z;
14:    int colsAdded = 0;
15:    //add derivatives to Jacobian:
16:    if Joint i has X DOF then
17:        AddColumn(J, dPx, dRx, colIndex);
18:        colsAdded = colsAdded + 1;
19:    if Joint i has Y DOF then
20:        AddColumn(J, dPy, dRy, colIndex + colsAdded);
21:        colsAdded = colsAdded + 1;
22:    if Joint i has X DOF then
23:        AddColumn(J, dPz, dRz, colIndex + colsAdded);
24:        colsAdded = colsAdded + 1;
25:    for j = i+1..n do
26:        //if the current link is higher up the chain than the link being evaluated for contacts:
27:        int rowIndex = 6;
28:        for k = 0..nc_j do
29:            colsAdded = 0;
30:            Contact c = c_ik;
31:            //compute contact derivatives:
32:            if Joint i has X DOF then
33:                dPCx = a'_i x × (p_c − p'_i);
34:            if Joint i has Y DOF then
35:                dPCy = a'_i y × (p_c − p'_i);
36:            if Joint i has Z DOF then
37:                dPCz = a'_i z × (p_c − p'_i);
38:            //add derivatives to Jacobian:
39:            if Joint i has X DOF then
40:                J[rowIndex, colIndex] = dPx.x;
41:                J[rowIndex + 1, colIndex] = dPx.y;
42:                J[rowIndex + 2, colIndex] = dPx.z;
43:                colIndex = colIndex + 1;
44:            if Joint i has Y DOF then
45:                J[rowIndex, colIndex + colsAdded] = dPy.x;
46:                J[rowIndex + 1, colIndex + colsAdded] = dPy.y;
47:                J[rowIndex + 2, colIndex + colsAdded] = dPy.z;
48:                colIndex = colIndex + 1;
49:            if Joint i has X DOF then
50:                J[rowIndex, colIndex + colsAdded] = dPz.x;
51:                J[rowIndex + 1, colIndex + colsAdded] = dPz.y;
52:                J[rowIndex + 2, colIndex + colsAdded] = dPz.z;
53:                colIndex = colIndex + 1;
54:            rowIndex = rowIndex + 3;
55:    colIndex = colIndex + colsAdded;
```

In the above algorithm n is the number of joints in the chain, $a'_i x$ is the local x-axis for joint i ($a'_i y$ and $a'_i z$ are the local x- and z-axes, respectively), c_{ik} is contact number k for joint i, and nc_j is the number of contacts for joint j. The resulting Jacobian matrix has the following structure (for one contact):

$$\left[\begin{array}{ccc|ccc|c|ccc}
l_1x, p_ex & l_1y, p_ex & l_1z, p_ex & l_2x, p_ex & l_2y, p_ex & l_2z, p_ex & l_3z, p_ex & l_4x, p_ex & l_4y, p_ex & l_4z, p_ex \\
l_1x, p_ey & l_1y, p_ey & l_1z, p_ey & l_2x, p_ey & l_2y, p_ey & l_2z, p_ey & l_3z, p_ey & l_4x, p_ey & l_4y, p_ey & l_4z, p_ey \\
l_1x, p_ez & l_1y, p_ez & l_1z, p_ez & l_2x, p_ez & l_2y, p_ez & l_2z, p_ez & l_3z, p_ez & l_4x, p_ez & l_4y, p_ez & l_4z, p_ez \\
\hline
l_1x, r_ex & l_1y, r_ex & l_1z, r_ex & l_2x, r_ex & l_2y, r_ex & l_2z, r_ex & l_3z, r_ex & l_4x, r_ex & l_4y, r_ex & l_4z, r_ex \\
l_1x, r_ey & l_1y, r_ey & l_1z, r_ey & l_2x, r_ey & l_2y, r_ey & l_2z, r_ey & l_3z, r_ey & l_4x, r_ey & l_4y, r_ey & l_4z, r_ey \\
l_1x, r_ez & l_1y, r_ez & l_1z, r_ez & l_2x, r_ez & l_2y, r_ez & l_2z, r_ez & l_3z, r_ez & l_4x, r_ez & l_4y, r_ez & l_4z, r_ez \\
\hline
l_1x, p_cx & l_1y, p_cx & l_1z, p_cx & l_2x, p_cx & l_2y, p_cx & l_2z, p_cx & l_3z, p_cx & l_4x, p_cx & l_4y, p_cx & l_4z, p_cx \\
l_1x, p_cy & l_1y, p_cy & l_1z, p_cy & l_2x, p_cy & l_2y, p_cy & l_2z, p_cy & l_3z, p_cy & l_4x, p_cy & l_4y, p_cy & l_4z, p_cy \\
l_1x, p_cz & l_1y, p_cz & l_1z, p_cz & l_2x, p_cz & l_2y, p_cz & l_2z, p_cz & l_3z, p_cz & l_4x, p_cz & l_4y, p_cz & l_4z, p_cz
\end{array}\right]$$

The 'top' part contains derivatives for the position and rotation of the end effector, and is identical to the matrix resulting from the original algorithm, as in Section 6.1.1. The additional derivatives for contact position are added as additional rows. The exact number of rows in the Jacobian is thus dependent on the number of contacts any one joint has influence on in the current configuration (within the current frame). The example matrix includes values for only one contact. For each additional contact, there would be another set of three rows, one for each of the three dimensions of the contact position.

6.3 Additional Techniques

Apart from the basic techniques outlined above, we also implement some additional functions to hopefully ameliorate some of the common problems of Jacobian IK. Firstly, we have the spring constraint used in conjunction with the pseudoinverse solver. The technique as outlined in Section 4.2.8 tries to reduce the method's inherent instability near singularities (which occur most commonly in our simulation when the limb is fully extended).

There is also the problem that joint clamping is only applied when the actual rotation occurs. The joint limits are enforced by the clamping, but not taken into account in the construction of the Jacobian itself. This may lead to suboptimal solutions, since the possible space of solutions the Jacobian solver uses is not identical to the actual one. Therefore, its 'optimal' solution may include rotations that violate the joint limits. A tentative attempt to solve this problem was made by excluding DOFs from the Jacobian if rotation around that DOF would result in a violation of the joint limit. Results of this experiment are shown in Section 8.

6.4 Summary

We discussed exactly how the Jacobian matrix containing derivatives for all degrees of freedom in the system is constructed in the BirthPlay application. The process was further explained using a pseudocode version of the algorithm used to construct the Jacobian. We also discussed collision handling, the shape of the collision bodies, how collisions are detected and recorded, and the collision response, which is to add values to the Jacobian matrix that represent a corrective impulse of the contact point. In the next section we will discuss some of the finer details of the implementation of IK in the BirthPlay application, including the controller input, skeleton model, and programming.

7 Implementation

This section will present some of the basic information about the BirthPlay application that is relevant to our implementation of the inverse kinematics methods. This consists of an explanation of the 6-DOF controllers and how they are used for input, and a basic explanation of the structure of the skeleton and joints. Finally, we discuss in more detail how the BirthPlay application is designed and programmed, and the tools and libraries we use.

7.1 Controllers

The controllers used in the BirthPlay application are Razer Hydra motion controllers, as seen in Figure 14. The controller set is produced and marketed as a gaming implement, and is comparable to the Nintendo Wii Remote and Playstation Move controllers.

Figure 14: The Razer Hydra controller set, and a close-up view of one of the controllers.[2]

The controllers also each feature 4 action buttons, an analog stick, a trigger and a bumper. The aim is to use these buttons to control the whole application, without the need for a keyboard or mouse. This serves to improve ease of use and keep the user focused on the operation.

To track the user's hand movements, the controllers use magnetic motion sensing. Then, the base station tracks and computes their exact location and orientation, so there is no additional computation required in our application. We simply get the data representing the position and orientation for each controller, each frame. This gives a total of six degrees of freedom for each controller. We use these same degrees of freedom as the target vectors for the IK algorithm, in order to map the user's movement to the virtual doctor's. In the BirthPlay application, this mapping is limited to the arms only (wrist, elbow, shoulder, and clavicle joints), in order to keep the simulation simple and easily overseeable. This also has the effect of limiting the virtual character's range of possible motions.
The only operations needed to map the controller data to the virtual environment are to scale the position values as needed and to determine where to place the world origin of the simulation. This last one is relevant because the world origin of the simulation correponds to the position of the controller base. Depending on which movements we want the user to make, the origin may need to be altered. Of course, we can also transpose the position data by a certain amount instead of moving the world origin itself, if that is preferable.

Since the controller data is the only data we have, we naturally have no idea of the position of any of the user's limbs or other body parts. The IK algorithm will therefore give a best-effort solution, complying with all joint rotation limits and following from the previous solution. This will hopefully effect a realistic and believable motion, if not one perfectly identical to the user's.

7.2 Skeleton

This section discusses some of the background information that is important to our understanding of the way the BirthPlay application handles articulated skeletons for 3D animation. While the actual construction of the skeleton system is not a focus of our research, it is nonetheless important to know exactly how we can make changes to the skeleton's which information is available to us in order to carry out our operations. First, we will discuss the skeleton's structure and how it is manipulated. Secondly, we will discuss the different joint types and their uses.

7.2.1 Structure of the Skeleton Model

The skeleton structure used in the BirthPlay application is one seen in many 3D animation and simulation applications. It consists of joints linked together in a hierarchical tree structure. There is one root joint, which is the only joint without any parent joints. All other joints are child joints of some other joint. A joint can have multiple child joints, of course, but only one parent. This is to keep the system simple. It prevents the simulation of a completely accurate model of the human body, since, for example, the two bones in the lower arm will be represented by a single parent-child link, but this makes no difference for practical purposes. As might be expected, tranformations done on any one joint also affect its child joints. A practical example would be a movement of the shoulder, which will also affects the position and orientation of the elbow, wrist, and internal hand joints. Because all other joints are child joints of the root, it is easy to manipulate the whole skeleton's position or orientation by manipulating the root only.

The bones themselves are modeled as the links between joints. Their only characteristic is the distance between the two joints, or bone length. The bones' orientation and start and end points are derived from the joints' orientations and positions.

7.2.2 Joint Types

The joint type describes the kind of constraints placed on the connected objects in a physical simulation. There are three basic types: the ball-and-socket joint, the hinge joint, and the slider joint (see Figure 15).

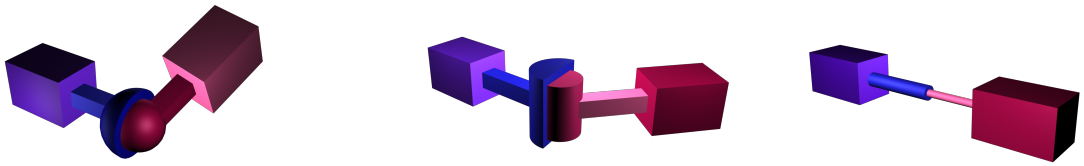

Figure 15: An illustration of the three joint types. From left to right: ball-and socket joint, hinge joint, and slider joint.[5]

The ball-and-socket joint has three DOFs, meaning it can rotate around all three local axes (x, y, and z). An example within the human body is the shoulder joint. The hinge joint limits rotation to one DOF, so it can only rotate around one local axis. Which one is used is arbitrary, a choice of implementation. An example of a hinge joint in the human body is the elbow. Finally, the slider joint has no rotational DOFs, but a translational one. This allows the connected parts of the rigid body to move along one of the local axes while keeping the same relative orientation. This joint type is mainly used in robotics applications, and is not implemented in the BirthPlay application.
Of course, though the ball-and-socket and hinge joints limit the amount of DOFs each joint has already, further constraints are needed for an accurate physical simulation. For this reason, rotation is limited by constraining each DOF to a certain region. This allows also for the simulation of joints with only two DOFs using the ball-and-socket joint, by simply constraining one DOF to no rotation at all. The joint

limits themselves are determined according to actual measurements of the human body. To enforce the limits, rotation is checked and clamped whenever the joint is manipulated.

7.3 Structure and Programming

This section discusses the structure and programming of the BirthPlay application as it relates to IK. It also names the external tools and libraries that were used for certain functions.

7.3.1 Class Structure

The BirthPlay application is written in the object-oriented C++ programming language. We will give here a brief overview of the class structure of the BirthPlay application's IK implementation. The application as a whole is of course a great deal more complicated, but its full description is not within the scope of this report.

As a whole, the IK methods are contained in the kinematics component of the BirthPlay system. This component contains three main classes: *Link*, *Joint*, and *Chain*. The *Link* class models a single link in a link chain. It has methods that are used from the *Chain* class to solve IK problems when 'local' calculations make the most sense, such as in the CCD method. Changes to the configuration are effected through use of the *Visitor* design pattern, which allows us to create a *Visitor* class that alters certain variables. There are *Visitor*s to read and write the relative and world positions of the *Link*, among others. An instance of the *Link* class knows its parent *Link*, relative position and orientation, radius, and *Joint*.

The *Joint* class is basically an extension of the *Link* class. Each *Joint* must have a *Link*, but the opposite is not necessarily true. Basically, the *Joint* contains specific methods needed to manipulate its position and orientation depending on its type. As was explained in Section 7.2.2, there are three types of joints: ball-and-socket joints, hinge joints, and slider joints. Each has its own types of limits and its own methods and variables. During normal function of the program, a *Link* will pass calls to manipulate it to its *Joint*, which makes the actual changes based on its type.

The *Chain* class is basically a collection of *Link*s arranged as a chain, with each *Link* being either the child or parent of another. Apart from knowing the base and end links in the chain, the *Chain* class is mainly used to call the actual IK methods. These methods are each contained in a single function. The *Chain* makes its calculations according to the chosen IK method, using its *Link*s' position and orientation data as needed. The calculated optimal changes to joint DOFs are then propagated to each *Link* in turn.

The IK methods are called at runtime on the relevant chains within each frame of the program. This is done using the *solveIK* method in the *Scenario* class. The *Scenario* class contains all variables and methods relevant to a specific practice scenario in the BirthPlay application. This includes the controller position and orientation data and the *Chain*s for each arm of the doctor model. At each frame, it calls these *Chain*s' IK methods to update their configuration within the orientation.

To quickly and easily construct new training scenarios and alter simulation values (such as joint limits, collision depth, etc.), simulation data is not hard-coded within the source code. Instead, we use script files written in the XML format, which are read at runtime. This enables us to update simulation parameters without the need for recompilation of the source code.

7.3.2 External Tools and Libraries

The BirthPlay application uses the Ogre open source 3D graphics engine for rendering [3]. Ogre uses vectors and quaternions to model position and rotation, making it easy for our implementation to interface with it. However, we do not use the Ogre classes directly in our calculations, but only for the rendering itself. Instead, we use the C++ MoTo (Motion Toolkit) template library for internal modeling of variables, including vectors, rotations (in axis-angle and quaternion notations) and associated

functions [16]. In this library, all variables are defined as templates, which makes it more flexible when definitions change, though that is not relevant to our implementation. A tool program is used to convert Ogre skeleton files into Skeleton objects that are used in the BirthPlay application.
To construct the Jacobian and do matrix manipulation in general, we use the armadillo linear algebra library [1]. This library contains classes that model matrices, as well as methods for all common matrix operations. The methods for the inverse and pseudoinverse in particular were useful for our project.

7.4 Summary

We discussed some of the basic information relevant to BirthPlay application's implementation of IK methods. This included the use of the Razer Hydra motion controllers, the basic skeleton structure, the three joint types (ball-and-socket, hinge, and slider), and an overview of the program structure. In the next section we will evaluate the program's performance, both for general IK and with regard to its handling of collisions.

8 Results

We conducted several experiments to evaluate the program's performance. The experiments consist of a visual examination of the motion and a convergence test for performance, each for several specific scenarios. We discuss the results of all experiments in turn. First, the tests done with no collisions to validate the basic working of the Jacobian IK algorithm, then the tests done with collisions, tests done to validate the use of joint limits in the Jacobian itself, and finally performance tests to determine the convergence time, both in iterations and actual time, of the different IK methods in different situations.

8.1 Experiments

The examination of the system's IK solutions is difficult, since it cannot be done through a simple, or even a complex, series of mathematical equations. Evaluation through comparison to motion capture data would also be possible, but since the shoulder positions of the obstetrician are fixed and we already have accurate coordinates for the wrist positions through the controller data, the only relevant new data would be the elbow positions. Since the elbow positions can vary significantly for the same motion without necessarily making the motion less realistic, we do not consider the discrepancy between the recorded elbow positions and the simulated positions valuable enough as a metric to devote time and resources to motion capture recordings.
The qualities we primarily value in the resulting animation are abstract. We want to know how realistic and believable the motion will look to a human user. The greater part of our examination must therefore be visual. Ideally the examination of the results would be carried out through independent testing by outside observers with no pre-existing knowledge of the system, but time constraints prohibit this. We therefore defined a set of testing scenarios that should theoretically cover all common operations within the BirthPlay application specifically. For this reason, there is only one testing environment where the different scenarios are carried out, which includes the virtual doctor, the mother, and the baby. Video footage of all tests has been recorded and will be made available for examination along with this report, since pictorial illustrations are unlikely to be sufficient. Fortunately, or perhaps unfortunately, the main flaws of the currrent implementation will be apparent to any observer through even non-extensive testing, and all will be highlighted over the course of this section.
We did not record footage of the user and controllers during our experiments. In all images and video footage of our experiments, there are two markers present consisting of three orthogonal axes in red, green, and blue. These markers represent each controller's position and orientation, which were used to reach the configuration of the doctor's arms visualized in the same frame.

We describe here the test scenarios used in our visual evaluation. The scenarios for the evaluation with no collisions are as follows:

1. Stable position, target parameters within reach.
2. Stable position, target parameters out of reach.
3. Slow movement, target parameters within reach.
4. Fast movement, target parameters within reach.
5. Rotation of the wrist which necessitates arm movement to accomodate the full rotation.
6. Retracting the wrist while the arm is fully extended.

The tests with no collisions were repeated with collisions to verify that the collision algorithm did not cause a difference in the results. The following are the additional scenarios used to evaluate situations specific to the collision handling:

1. Stable position, target parameters out of reach due to obstruction by the collision object.
2. Moving the target into a position behind the collision object, causing a collision.
3. Moving the target from a position behind the collision object back to collision-free space.

8.1.1 Experiments without Collisions

We start our testing without the inclusion of collision handling, to evaluate the basic performance of the Jacobian IK method. The collision handling carries with it a great deal of complications in the algorithm and its resolution, so we prefer to first ensure that the inverse kinematics part of the algorithm works as expected. A visual examination was conducted for each of the three 'inversion' methods (Jacobian Transpose, Jacobian Pseudoinverse, and Damped Least Squares). Furthermore, we also examined the Jacobian Pseudoinverse method's results with the use of the spring constraint to aid in case of singularities. Footage of these experiments is available accompanying this report. We discuss here only the general performance and any outliers or exceptions that we observe.

The most significant case of unrealistic motion is that in which the target position is out of reach when using the pseudoinverse method, in particular without the use of the spring constraint to aid in handling singularity cases. As can be clearly seen in the top-right part of Figure 16, this configuration causes such instability that the bones jump around in and out of all sorts of configurations. The pseudoinverse method *with* the spring constraint (bottom-left) performs better, as it is more stable. However, it is also clearly not an optimal solution. The Damped Least Squares method (bottom-right) also shows a somewhat suboptimal solution. However, it is so much closer to the target that in practice the difference might not be noticeable. The controller positions will normally be close to or within reach relative to the positions of the virtual doctor's wrists. Therefore, the slight discrepancy that occurs when the target is out of reach may not be problematic.

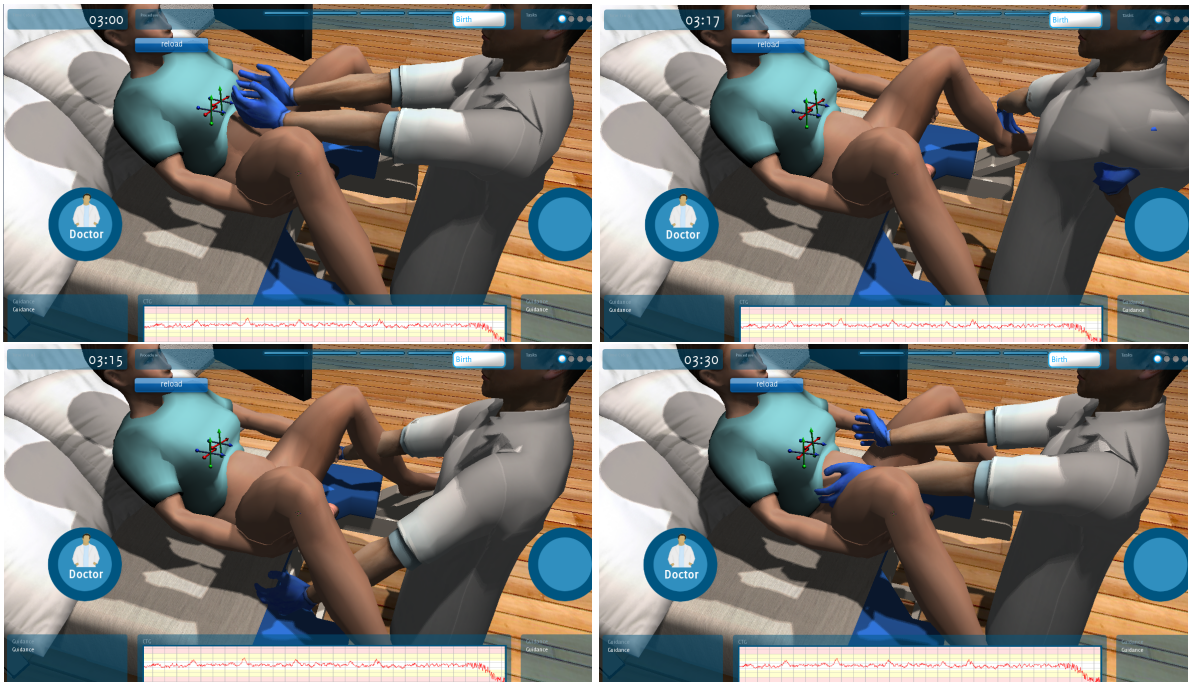

Figure 16: A stable controller position out of reach of the virtual model. From left to right: Jacobian Transpose, Pseudoinverse, Pseudoinverse with spring constraint, and DLS.

Under normal movement with no collisions, most algorithms seem to perform as well as can be expected (except for the previously mentioned situation where the target position is out of reach). The movement of the wrist follows the controller's target position closely and converges quickly enough that there is no significant visible delay. Joint limits are also observed at all times, and there is little instability. The exception for movement with the target within reach is the pseudoinverse method without spring constraint, which suffers from instability and jitter. This is likely due to the nature of the Jacobian, which is only valid near the current configuration. Therefore the solution becomes worse when the changes in DOF values are too large. With the addition of the spring constraint the valid solution space becomes

larger, which makes the movement more stable as expected.

Though the motion does lag behind the controller a bit for the other algorithms it does not come across as unnatural, because it converges quickly enough. If there is one observation that most readily springs to mind, it is that the rotation of the controller is translated to the model quicker than its change in position. In other words, the proper orientation seems to have greater priority than the proper position, which results in a virtual motion that looks unnatural. It even seems that the gap in position between the wrist and the target actually gets larger before converging. This is a result of the difference in weight between the two targets being solved for, those being the position and rotation of the controller, though strictly speaking each consists of three targets, one for each DOF. The weights of the different position DOFs and the different rotation DOFs are kept equal, however. This problem is most pronounced when the change in position is small, but the change in orientation is great. Otherwise, it is not as noticeable. The effect was observed for IK methods, though it was most pronounced for the Jacobian Transpose method. The rotation weight could be lowered, but this would result in such slow rotations that the user could not quickly turn his/her hands, creating new problems. Even finely tuning the weights cannot give good results for every configuration, since changes in position and orientation will have very different magnitudes at times. It would be better for the resulting motion if the rotation and position weights were dynamically altered to fit the situation. This could be done by altering the weights according to the relative difference in magnitude between each constraint's discrepancy between the current configuration and the target.

It can also be observed from tests involving a movement of the wrist that necessitates movement of the whole arm to accomodate the new wrist position that the optimal solution in regards to difference in position and rotation as a whole is sometimes not as natural as a suboptimal one.

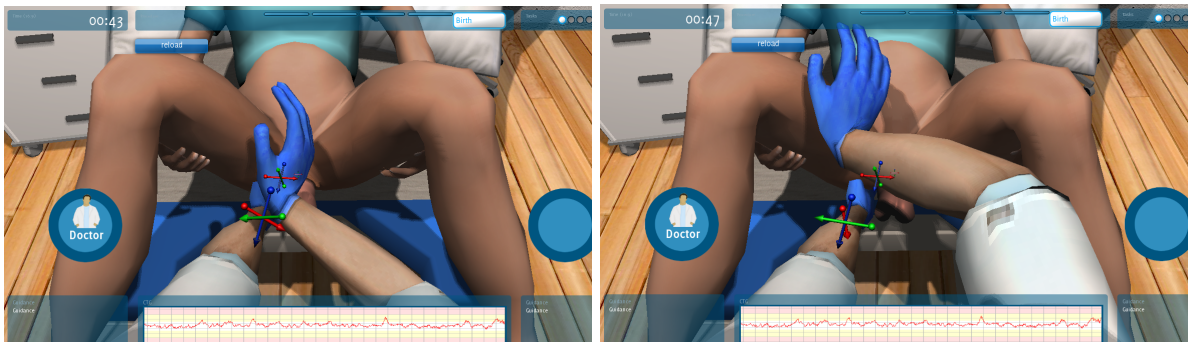

Figure 17: An illustration of an unnatural optimal solution.

Figure 17 illustrates one such scenario. The unnatural configuration is caused by the adherence to the joint limit of the wrist, which prevents it from rotating to match the controller's orientation. The optimal balanced solution between rotation and position results in the configuration on the right, which looks unnatural and uncomfortable (though technically possible, since no joint limits were violated), as well as being at a significant distance from the desired position and orientation. This type of situation will of course not occur often, since the joint limits of the human operating the controllers are presumably identical to those of the virtual doctor, and their positions should also be similar. Since all joint limits were taken from biometrical data and can be considered absolutely correct, altering the wrist joint limit itself would not be a proper solution. Since configurations where the maximum possible rotation is reached are usually naturally uncomfortable, it might help the solution to add a preference to solutions that are as far away from those limits as possible (within reason, of course). This might make for a more natural-looking solution. However, since our current solution to take the joint limits into account within the Jacobian itself results in instability (see also Section 8.1.3), we cannot offer any ideas on how to approach this type of solution.

8.1.2 Experiments with Collisions

When dealing with collisions, we immediately run into more severe issues than those that occured when dealing only with the basic Jacobian inverse kinematics. We see immediately that the end effector's movement stops as it collides with the collision object. However, to prevent this collision, the weight given to the contact effector's movement was increased so severely that the position and rotation of the end effector at the wrist are given almost no influence on the final solution. This is not an issue of fine-tuning the weights, as we observed collisions occurring when the weight was lowered even by a small amount. This manifests itself as very slow convergence towards other solutions as soon as the contact occurs, as seen in Figure 18.

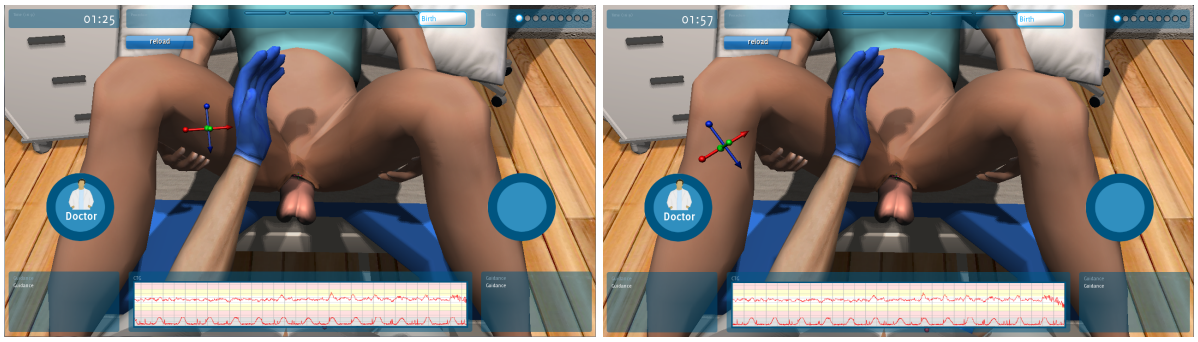

Figure 18: An illustration of convergence issues when colliding. The movement correctly stops when the collision occurs (left), but stops and does not converge to a solution closer to the target as the target's position changes (right).

The user can move the controller back to a non-colliding position and orientation, upon which the wrist snaps free, and the user can attempt to reach some other position before a contact again occurs. However, this problem cannot be solved using a weighted approach to the problem. A reduction of the contact effector's weight results in configurations where the doctor's model moves through the collision object. Even if the contact is stopped in certain situations, such as in a head-on, straight collision, a contact violation may still occur when the position and/or rotation targets are at a significant distance from the current configuration. It is not possible to prevent the collisions except by a drastic difference in weights.

8.1.3 Joint Limits in the Jacobian

In Section 6.3 it was explained how degrees of freedom could be selectively removed from the Jacobian within one frame to enforce the use of joint limits in the Jacobian itself. However, we can conclude from our tests that this method does not result in a noticeable difference in the resulting motion, and in some cases can result in a jittery effect on the wrist's movement, where it rapidly alternates between positions that are close together.

The cause is probably that the altered Jacobian gives a different result for the optimal position, but since the position is different in the next frame, this also influences the Jacobian, which then gives a different optimal solution, and so on. The solution of removing a degree of freedom altogether may thus be too drastic, and a different solution should be found to limit its range without completely removing it for better results. This could theoretically be done through additional constraints within the Jacobian. However, we can offer no ideas on how to express these constraints in practical terms.

8.1.4 Performance

Though most of the evaluation necessarily must be visual, there are still certain parts of the results that are best measured numerically. The performance of the IK algorithm is an important part of the evaluation, since the BirthPlay application is absolutely required to work properly in real-time. It would therefore be unacceptable if the algorithm took too long to converge on a proper solution. To evaluate the algorithm's performance in this area, we conducted tests to record both the amount of iterations

and the actual time needed for convergence. These results are for the complete convergence into the solution, regardless of frame numbers. The moment of convergence in these tests is considered to be that in which either the difference between the desired position and orientation of the end effector and its actual position and orientation is sufficiently negligible, or that in which the difference between the position and orientation of the end effector in the previous and current iteration is sufficiently negligible.

The test scenarios are as follows:

1. No collisions, a movement where the arm is retracted from an extended configuration. See Figure 19.
2. No collisions, a movement where the arm must extend to the far left or right. See Figure 21.
3. Collisions active, a movement where the arm moves into a contact position. See Figure 23.
4. Collisions active, a movement where the arm moves from a contact position to a collision-free position. See Figure 25.

We provide an illustration of each test, along with graphs which give the results for the different Jacobian IK methods for both iterations and time.

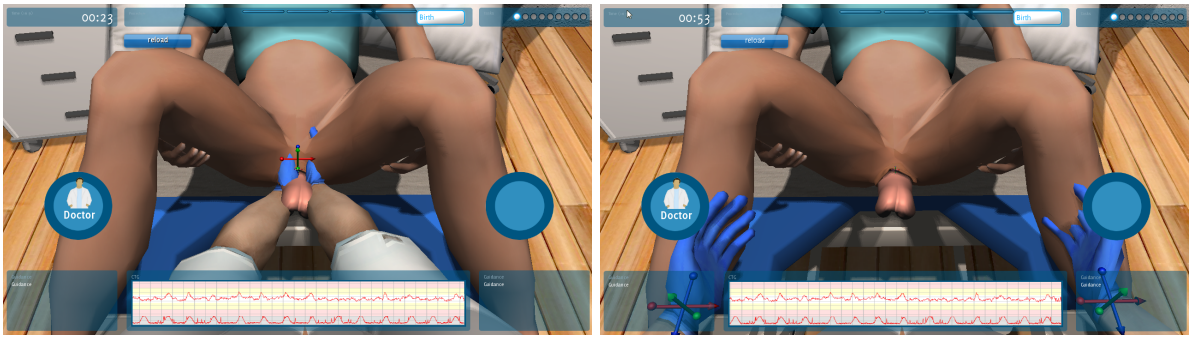

Figure 19: Test 1: a movement where the arm is retracted from an extended configuration.

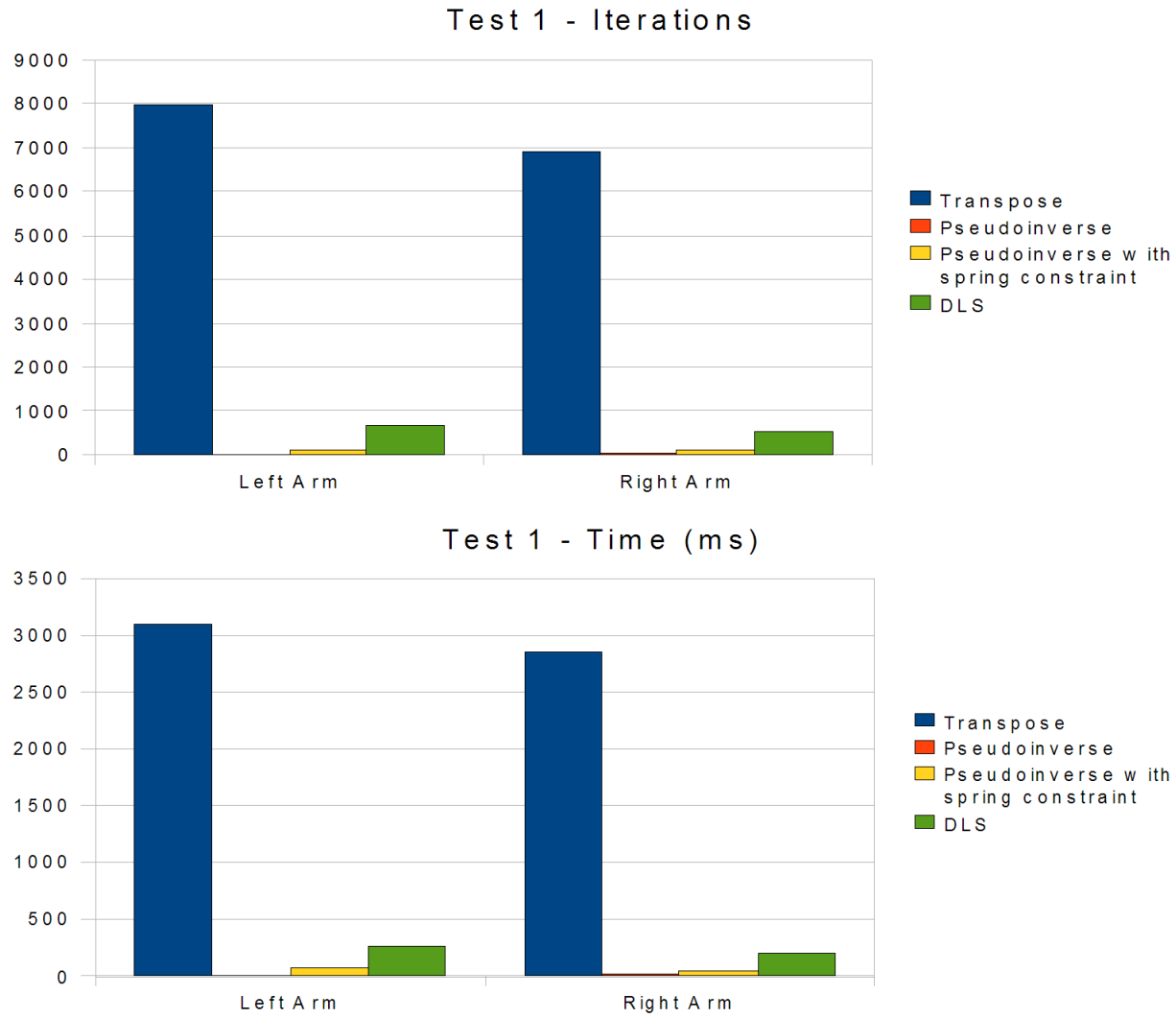

Figure 20: Graphs for iterations and time for test 1.

Test 1 illustrates how slowly the transpose method converges compared to the pseudoinverse and DLS methods. The pseudoinverse method converges the fastest by far, with the spring constraint adding some iterations and time, as could be expected due to the space it provides for valid solutions around the actual target. For the same reason, the DLS method also converges more slowly. The difference between the pseudoinverse method's results and the other methods' is at its most pronounced here. In fact, so is the difference between the transpose and DLS. This is likely because the movement of retracting the extended arm requires a greater exactness in computing the inverse, as it requires great changes in the DOFs of the joints in the arm, most notably the elbow and shoulder. The pseudoinverse is of course the best approximation of the actual inverse, and the transpose the worst, so the difference here is logical.

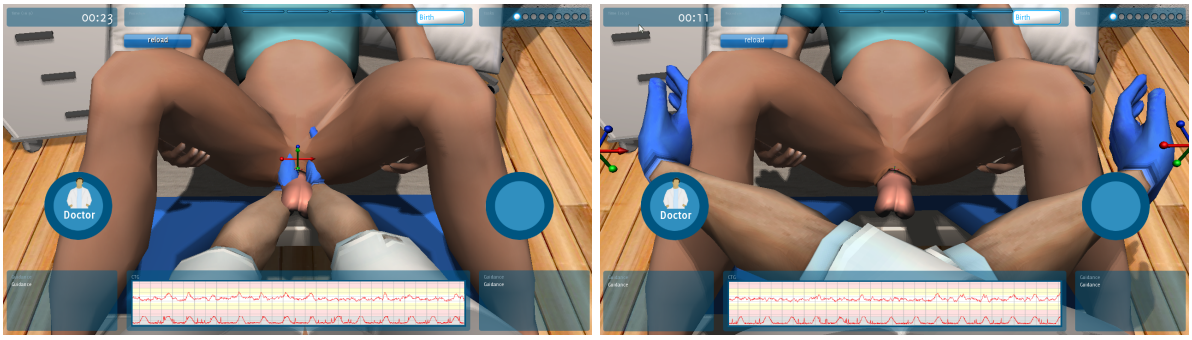

Figure 21: Test 2: a movement where the arm must extend to the far left or right.

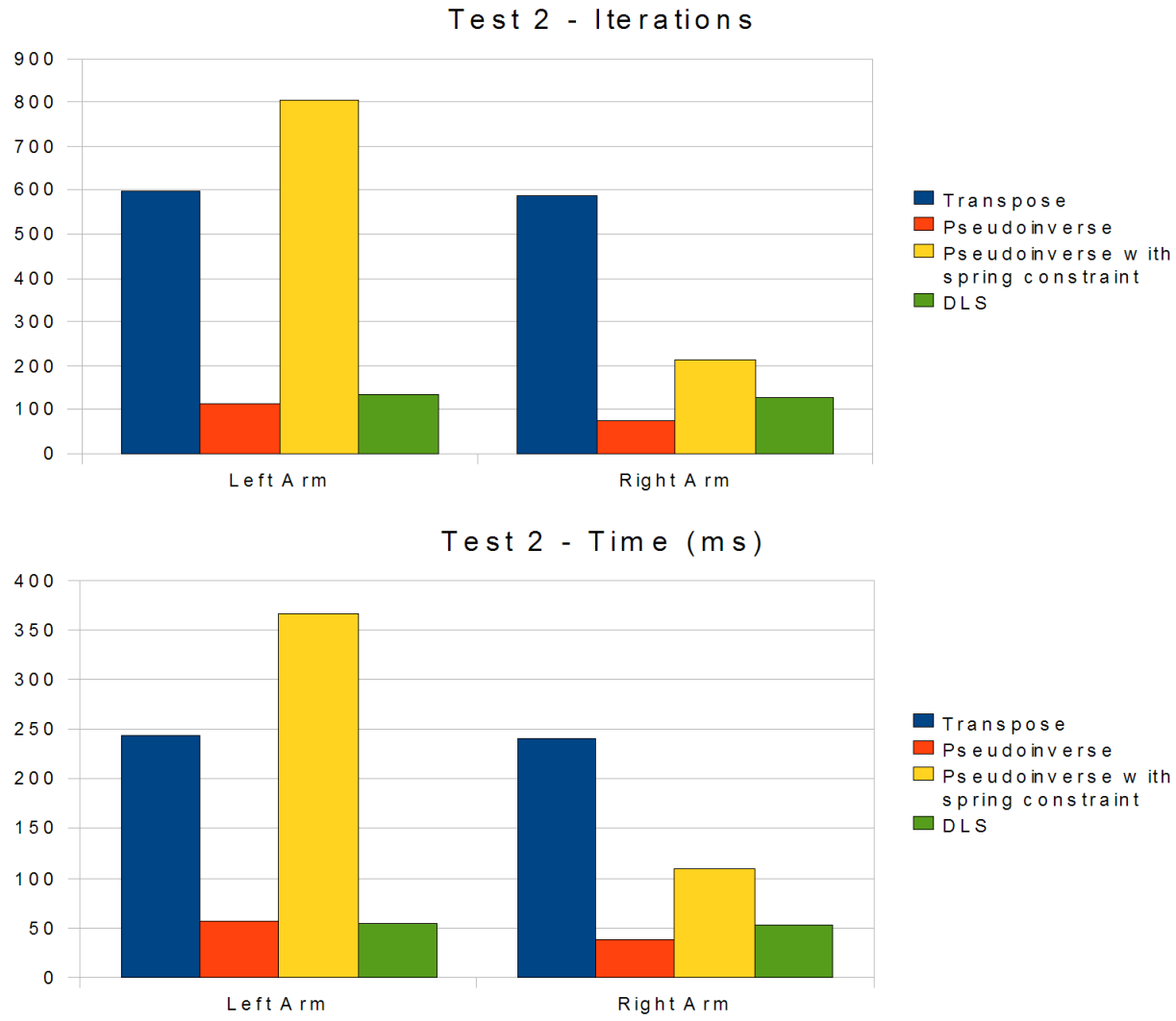

Figure 22: Graphs for iterations and time for test 2.

For test 2 the outlier is the pseudoinverse with spring constraint method. The left arm needs four times as many iterations and time to converge properly. Further examination of the raw data indicates that even when it has converged to the proper position, it still needs two iterations each frame to reach a solution. Visually, there was no instability in the resulting configuration. The cause is likely that it runs into a joint limit, so that it takes a step forward in the first iteration, but then sees no change in position

and rotation in the second, and goes to the next frame without actually changing the configuration. We also see that the difference between the methods is smaller than in the previous tests, probably for the reason outlined there, that the convergence there required a greater change in DOFs and thus more calculations, and a greater importance was placed on the accuracy of the approximation of the Jacobian inverse. The difference between the pseudoinverse methods and DLS is also smaller for this reason.

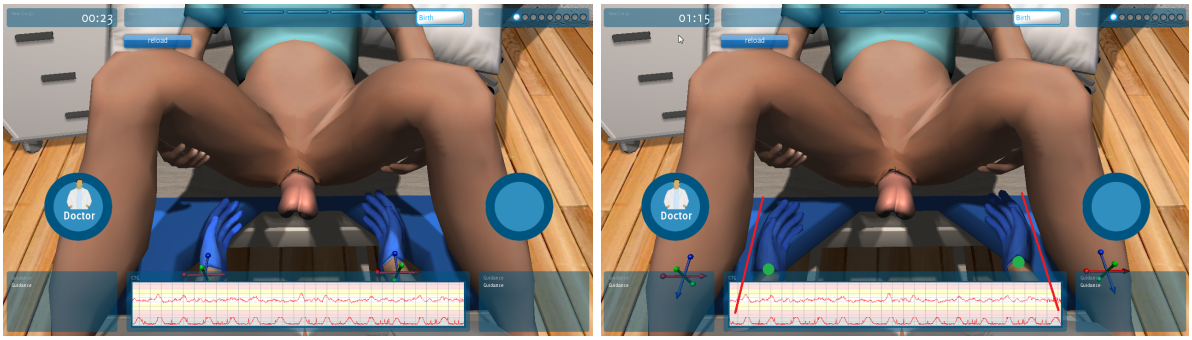

Figure 23: Test 3: a movement where the arm moves into a contact position. In the right picture, the contact is marked by the red line, the end effector by the green dot.

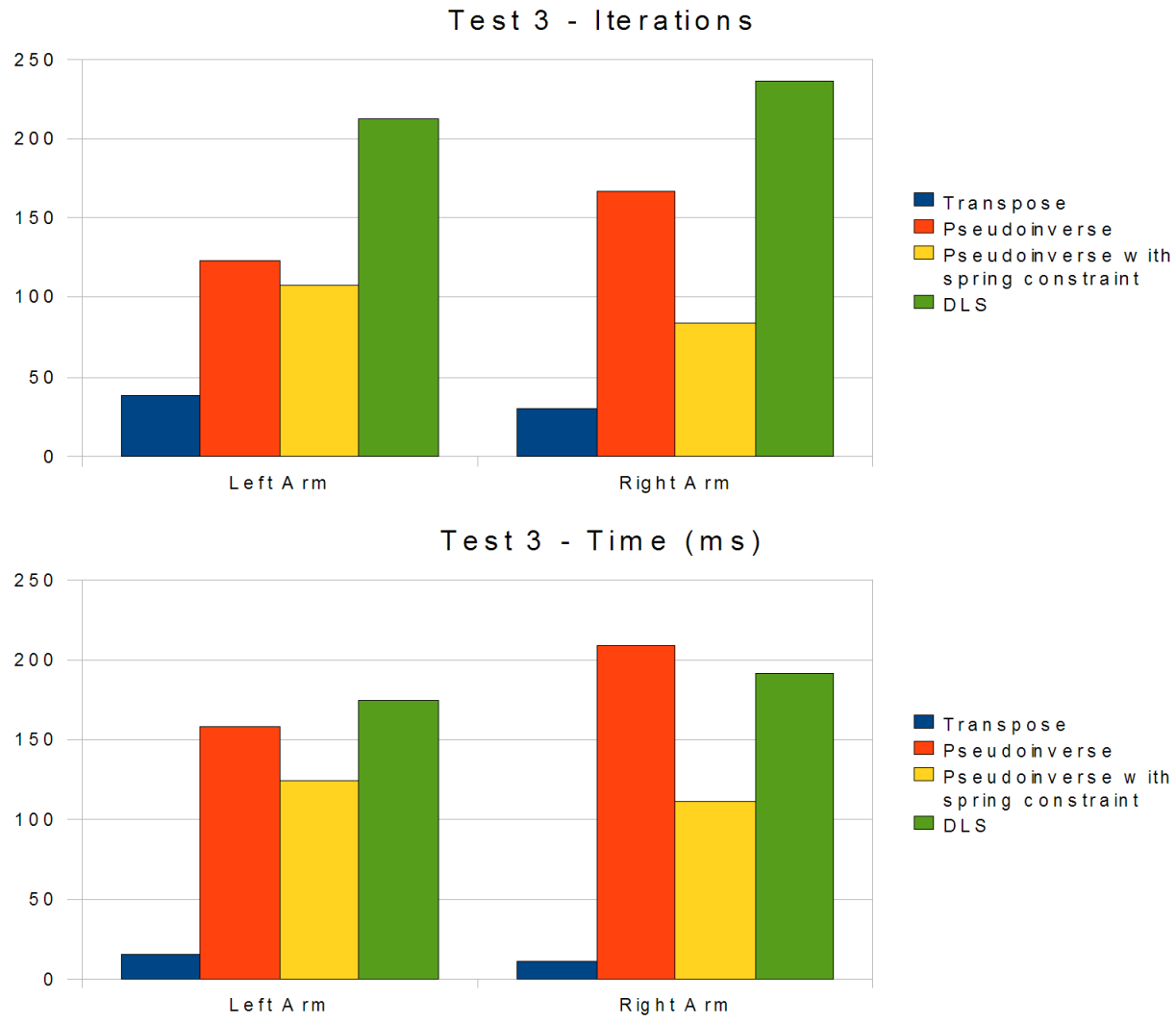

Figure 24: Graphs for iterations and time for test 3.

With test 3 we see a great reduction in the number of iterations and time (logical, since the movement is such a small one). We also see that the transpose method is now much more efficient than the others, with DLS replacing it as least efficient method. The normal pseudoinverse method is also less efficient in this case than the pseudoinverse method with spring constraint. This may be due to the added collision constraints restricting the solution space so that the pseudoinverse method must take more iterations and time to converge exactly. The addition of the spring constraint then allows for quicker iteration towards the solution by enlarging the solution space, offsetting the extra computations needed to add the spring constraint.
In this case the difference between the right and left arms cannot be explained by the joint limit, as above, since the data does not support it. It simply took more time for the right arm to converge than the left. This suggests that there is some asymmetry in the pseudoinverse solver, but it may also be due to the properties of the collision shapes and geometry of the doctor character, which we both know to be not perfectly symmetrical.

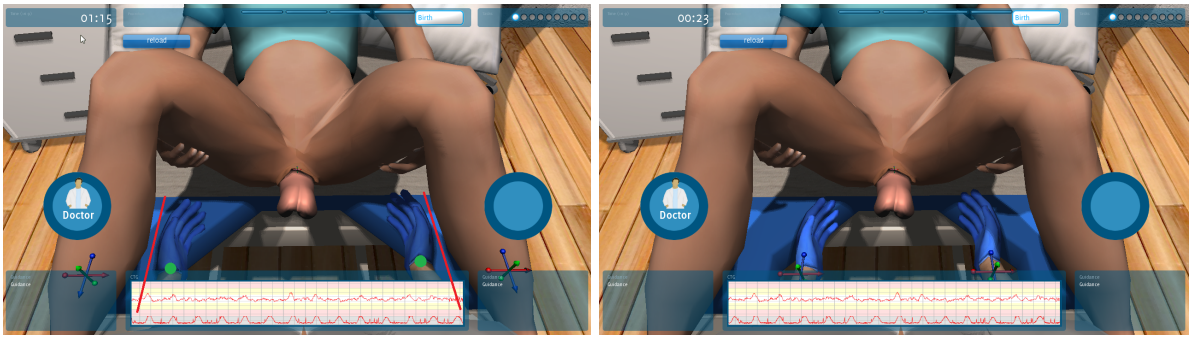

Figure 25: Test 4: a movement where the arm moves from a contact position to a collision-free position. In the left picture, the contact is marked by the red line, the end effector by the green dot.

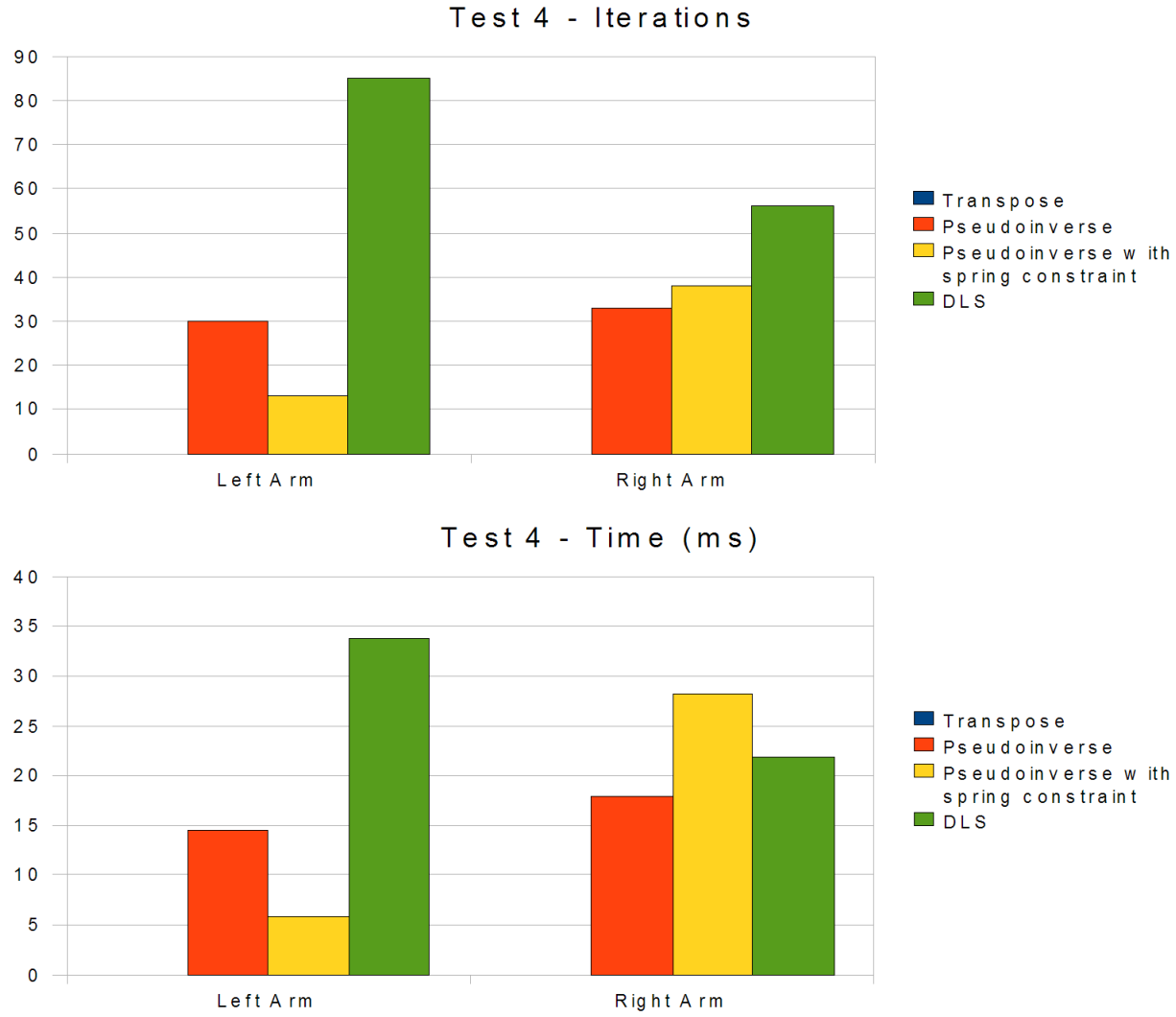

Figure 26: Graphs for iterations and time for test 4. The Jacobian transpose method did not converge and so has no results.

Test 4 has no results for the Jacobian transpose method, because that method did not converge on the target position for each wrist, and instead remained in contact position. The difference between left and

right arms this time is more easily explained as due to asymmetricalities and inaccuracies in the system and calculations. In any case the scale of the number of iterations and time is so small it is hard to say anything definite about this data. That the transpose did not converge is relevant, however, as it illustrates a limitation of that method's simple way of finding the approximation of the inverse. Evidently a more accurate approximation was necessary to find the proper way to return to a contact-free position. On the other hand, a deadlock such as this would not occur if it were not for the way the 'weighted' approach to priority is implemented in this system, so it might not be a problem when using a different approach.

8.2 Summary

We discussed the results of our implementation of Jacobian IK in the BirthPlay application. The general IK worked as expected, including the difference between the results of the different methods (Pseudoinverse, Pseudoinverse with spring constraint, DLS, and Transpose), but the collision handling method we used could be improved. For the visual results, all methods performed well with no collisions, but the pseudoinverse method becomes unstable when the target is out of reach, and even with the spring constraint does not give a realistic result. With collisions, all methods had significant problems with convergence, so that none could be considered at all realistic. Performance-wise, the transpose method had the worst results on the whole. The pseudoinverse methods had the quickest convergence, especially for large movements. The difference between the pseudoinverse method with spring constraint and without favors the one without, but in view of the visual results the method with spring constraint seems better. The DLS method is on the whole slower than the pseudoinverse methods, but still seems to perform well enough for real-time application and has the least visual instability. In the next section we will formulate our conclusions as to the program's performance and areas in which the method might be improved in future work.

9 Conclusions and Future Work

9.1 Conclusions

In this project, we formulated a system for handling inverse kinematics using an approximation of the Jacobian inverse. The system works well for most general motions, both for changes in position and orientation. Solving for orientation is done using the whole chain rather than just the end joint, which is an improvement over the previous implementation. We also supplemented the existing clamping solution for joint limits with the removal of degrees of freedom in the Jacobian matrix, though this ultimately causes too much instability in the solution to be useful. The spring constraint added to the Jacobian Pseudoinverse method has improved its results for configurations with singularities without otherwise impairing its performance.

However, we can conclude from the results in the previous section that the current implementation of Jacobian inverse kinematics in the BirthPlay system still has some significant flaws, most evident in its handling of collisions. It is clear from the results that the weighted approach to collision constraints does not work for our purposes, since the weight has to be too heavily tilted towards the collision constraint, so that it heavily slows or even prevents convergence. Also, for rotation and position, the fixed weights that are used right now sometimes create an unbalanced motion, with a great rotation that actually causes the difference in position between current and target configurations to become larger, which looks unnatural.

However, we are confident that the basic IK system fulfills its function well, producing adequate approximations of human motion from the input data, and should thus be a good basis to build on with a better collision handling system. The three Jacobian IK methods have been tested in different respects, both with and without collisions. The Jacobian Transpose method gives good results for small and simple motions close to the current configuration. However, it converges much slower when the target is further away and has the most trouble of the three when a good approximation of the Jacobian inverse is key. The Jacobian pseudoinverse method does much better in terms of efficiency in all situations. However, it has trouble with the singularities caused by fully extended limbs, making it of little use when the target is out of reach. The addition of the spring constraint does increase its stability, but the resulting solution is still noticeably off. Of the three different Jacobian IK methods, we give preference in the current system to the Damped Least Squares method for its stability and reasonable convergence rate in all test scenarios.

9.2 Future Work

The main conclusion of our tests is the insufficiency of the 'weighted' approach to priority when dealing with collision constraints. Future development of the BirthPlay system should find a better way to approach this issue. A possible solution could be to implement priority levels for IK constraints and collision handling, so that collision constraints can be solved first and guaranteed to be satisfied before dealing with the actual position and orientation of the user's hands. The system proposed by Baerlocher et al. [7] may be useful in formulating an approach in this direction.

If for some reason a system based on priority levels cannot be achieved, perhaps because of constraints on available time for calculation of solution spaces, the existing weight structure still has some room for improvement. We found that constant and fixed weights were undesirable in some situations, so a system with dynamic weights depending on the current configuration and goals, as well as priorities, could be an improvement. Such a system would likely still not work for collision handling, but its basic IK functionality might look more realistic and believable on the whole.

Additionally, the way that joint limits are currently handled in the Jacobian should be improved. Instead of creating a more natural motion, it creates, in some configurations, a jittery effect in the end effector that harms the believability of the animation. A better solution may be to limit the range of movement instead of removing the DOF altogether. However, this is not trivial, and we can offer no indication as to a way to go about it here.

Finally, there are some ways in which the system as a whole might be improved to make the simulation more lifelike. Currently, the doctor can hold the baby using several different 'grips'. These grips are predefined, as is the animation of the doctor's hands as it grasps the baby. It might be more realistic to use inverse kinematics to move the fingers instead, using individual target parameters for each finger. This would also allow for dynamic alteration of the grip as the baby model itself is animated and its collision shapes move, though defining optimal target parameters and finding a balanced solution may involve time-intensive calculations. The same technique could be used to allow the doctor to interact with other bodies or objects.

The system also currently makes no allowance for physical characteristics of actors and objects. There is no consideration for collision speed, impact, or the weight and substance of the various bodies and objects in the simulation. A physics model could improve the realism of the simulation and in particular its collisions. However, this addition might not be viable if it leads to instability or a significant increase in computation times. This is because real-time interaction between the user and the simulation is an important part of the system.

References

[1] Armadillo c++ linear algebra library. `http://arma.sourceforge.net/`. Accessed: 2013-11-12.

[2] Razer hydra controller. `http://www.razerzone.com/gaming-controllers/razer-hydra/`. Accessed: 2013-09-29.

[3] Ogre open source 3d graphics engine. `http://www.ogre3d.org/`. Accessed: 2013-11-12.

[4] Illustration of gimbal lock. WikiMedia Commons, 2009.

[5] Illustration of joint types. WikiMedia Commons, 2009-05-04.

[6] N.I. Badler, K.H. Manoochehri, and G. Walters. Articulated figure positioning by multiple constraints. *Computer Graphics and Applications, IEEE*, 7(6):28–38, 1987.

[7] P. Baerlocher and R. Boulic. An inverse kinematics architecture enforcing an arbitrary number of strict priority levels. *The visual computer*, 20(6):402–417, 2004.

[8] M.E. Bartelink. Global inverse kinematics solver for linked mechanisms under joint limits and contacts. Master's thesis, Utrecht University, 2012.

[9] S.R. Buss. Introduction to inverse kinematics with jacobian transpose, pseudoinverse and damped least squares methods. *University of California, San Diego, available from* *http://math.ucsd.edu/~sbuss/ResearchWeb/ikmethods/index.html*, 2004.

[10] S.R. Buss and J.S. Kim. Selectively damped least squares for inverse kinematics. *Journal of Graphics, GPU, and Game Tools*, 10(3):37–49, 2005.

[11] E. Catto. Soft constraints: Reinventing the spring (gdc 2011). *Game Developers Conference 2011, available from* *https://box2d.googlecode.com/files/GDC2011_Catto_Erin_Soft_Constraints.pdf*, 2011.

[12] R. Kulpa, F. Multon, and B. Arnaldi. Morphology-independent representation of motions for interactive human-like animation. In *Computer Graphics Forum*, volume 24, pages 343–351. Wiley Online Library, 2005.

[13] A. Liegeois. Automatic supervisory control of the configuration and behavior of multibody mechanisms. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(12):868–871, 1977.

[14] A.A. Maciejewski and C.A. Klein. Obstacle avoidance for kinematically redundant manipulators in dynamically varying environments. *The international journal of robotics research*, 4(3):109–117, 1985.

[15] C.B. Phillips and N.I. Badler. *Interactive behaviors for bipedal articulated figures*, volume 25. ACM, 1991.

[16] G. Van den Bergen. Dtecta. `http://www.dtecta.com/`. Accessed: 2013-11-12.

[17] G. Van den Bergen. A fast and robust gjk implementation for collision detection of convex objects. *Journal of Graphics Tools*, 4(2):7–25, 1999.

[18] C.W. Wampler. Manipulator inverse kinematic solutions based on vector formulations and damped least-squares methods. *Systems, Man and Cybernetics, IEEE Transactions on*, 16(1):93–101, 1986.

[19] L-C.T. Wang and C.C. Chen. A combined optimization method for solving the inverse kinematics problems of mechanical manipulators. *Robotics and Automation, IEEE Transactions on*, 7(4):489–499, 1991.

[20] C. Welman. *Inverse kinematics and geometric constraints for articulated figure manipulation.* PhD thesis, Simon Fraser University, 1993.

[21] J. Zhao and N.I. Badler. Inverse kinematics positioning using nonlinear programming for highly articulated figures. *ACM Transactions on Graphics (TOG)*, 13(4):313–336, 1994.