Department Of Computer Science, University of Bath
EngD in Digital Entertainment

Inverse Kinematics Coursework

CM50244 Computer Animation and Games I

Garoe Dorta Perez, Dave Hibbitts, Ieva Kazlauskaite, Richard Shaw
Unit Leader: Prof Phil Willis

November 21, 2014

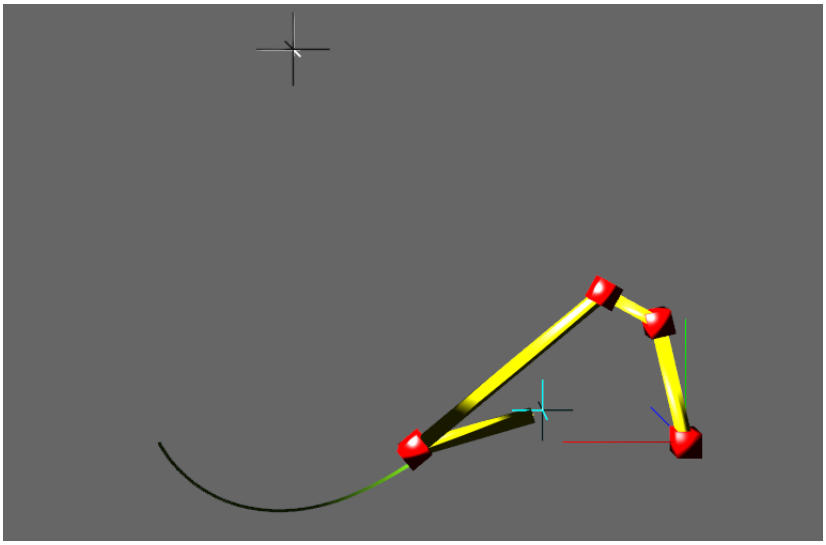

Figure 1.1: Final three-dimensional simulation in OpenGL, complete with the trace showing the motion of the end-effector, and a star signifying the goal position.

Abstract

In this project we explore the application of inverse kinematics to the control of a linkage in three-dimensional space. First, we construct a simple composite object which contains a chain of rigid rods of user-selectable lengths, that are connected by joints so as to allow full flexibility of movement. In the current implementation, the joints connecting the rigid rods can be restricted to either rotate purely in a particular two-dimensional plane in three-dimensional space, acting like simple hinges, or enabled to move freely in three-dimensional space, resulting in more complicated motions. The joints have the ability to be constrained to only move within user-defined limits, and the flexibility and rotation speed of the joints can be varied. Second, the method of inverse kinematics is explored in order to determine the appropriate configurations of the linkage at each time-step, i.e. the required angles of the joints when the linkage is given the task of reaching a user-specified target point. The method works by planning the motion towards a goal, which is defined by point in three-dimensional space, subject to constraints that ensure the target point is in a reachable region of the linkage; dependent on the overall length of the linkage and the maximum and minimum angles of the joints. Several commonly employed inverse kinematics techniques are discussed and evaluated, including nonlinear optimisation methods, but in the final implementation we utilise the Jacobian transpose method due to it's relatively simple implementation and fast computational speed, making it an attractive method for real-time and interactive applications such as this. Finally, the movement of the object is animated, and a number of both physical and non-physical parameters can be adjusted to deliver a visually appealing performance, for example by enforcing a slow-in and slow-out motion.

1 Introduction

Kinematics refers to the methodological study of movement, encompassing all the geometrical properties of motion. The motion or trajectory of an object in space can be described using either forward or inverse kinematics. The former uses known angles between parts of the object to determine the coordinates of the end-effector, while the latter uses a known final position of the end-effector to calculate the angles between the parts which ensure that the object reaches this desired target position.

Whilst the subject of inverse kinematics is applicable to many diverse areas, including astrophysics and robotics, this project is primarily concerned with its application to the fields of animation and computer graphics. And despite its prevalence is many different fields, inverse kinematics remains an interesting and difficult problem to solve. Inverse kinematics has garnered significant research and interest because it is much more intuitive than forward kinematics, especially for animators, where it is more desirable to define an end position and work out the required angles to get there. Yet it is challenging because many solutions can exist.

This report begins with a brief account of the main objectives of the project and then proceeds to discuss previous related work, including an overview of some of the most commonly used methods of approaching the inverse kinematics problem. Following this, the procedure of our investigation is explained, describing the different methods that were explored. Moreover, some of the issues that were encountered are discussed, including problems concerning both the computational and the implementation aspects, and how these problems were addressed. We then go on to discuss the simulation implementation of the proposed linkage system in Matlab, and the computational methods that were explored for solving the inverse kinematics problem. Carrying out an initial simulation in Matlab enabled us to be able to set up the equations quickly, test out different methods, and discover what worked and what didn't in a simplified two-dimensional system before porting the code to the final, fully three-dimensional environment in OpenGL. We then continue by explaining in detail the operations and execution in OpenGL. The last section of this report includes a review of the performance and the limitations of the system, as well as a discussion of possible further improvements.

1.1 Objectives

The main objective of this investigation was to explore the problem of inverse kinematics using a linkage in three-dimensional space. The final system was to be built in OpenGL and allow for a reasonable degree of user interaction. The linkage model did not need to exhibit a high level of realism, so long as the three-dimensional nature of the linkage was revealed, using simplistic lighting and rendering. The main properties of the inverse kinematic system to be explored were:

1. To implement a user-interactive way of choosing the target position for the end-effector of the linkage to reach in three-dimensional space.
2. A way to change the physical properties of the linkage, such as the rate at which the joints can move, any specified constraints on the joints, and the slow-in and slow-out movement of the linkage as it leaves it's current position and approaches the target position respectively.
3. To investigate whether to model and thereby vary the flexibility of the rods.

2 Related Work

The subject of inverse kinematics has been much researched over the past 60 years. Initially, it was primarily used in the robotics industry, for example for the control of robotic arms, particularly in car manufacturing, and the problem was first formulated in mechanical engineering literature [3]. But it has since been adopted by the computer graphics community as a means for animating a character's pose and movement through the control of an underlying moveable skeleton. An animator will often adjoin a three-dimensional deformable model of a character to an articulated skeleton structure, with defined bones and joints with certain physical constraints, which can be moved interactively into the desired positions. This process is known as 'rigging' in animation. The issue that inverse kinematics attempts to solve is to find the set of joint configurations that produce a plausible skeleton or linkage configuration based upon a desired

end-effector position. For example, in character animation, the end-effector may be the position of a hand or foot, and we want to know the required joint angles of the arm or leg which allow the end-effector to be in this position, whilst remaining physically plausible for the character.

There are a number of well-established methods of approaching the inverse kinematics problem which can be divided into four main categories of solver: geometrical/analytical algorithms, cyclic coordinate descent (CCD), differential algorithms, and hybrid methods which combine together various aspects of the first three techniques [6]. Geometrical and analytical algorithms, such as by [5], [2] tend to be very quick because the inverse kinematics problem is essentially reduced to a single mathematical equation that only needs to be evaluated in a single step to produce a result. However, such methods have limitations, particularly for linkages with a large number of links, and so in these situations it is impractical to reduce the problem to a single-step mathematical equation. This means that in the field of character animation, geometrical/analytical techniques are less useful, but for a project such as this, where we are dealing with only a few links and a single end-effector, this could perhaps be an appropriate method.

Another approach is cyclic coordinate descent (CCD), which uses a greedy algorithm to solve for each joint in turn, and repeats this process until a satisfactory solution is found. Solvers based on this method include those by ... Although this method is relatively simple and fast, it suffers from inaccuracies. Another issue with this technique is that the joint angles are updated one at a time, which has the undesirable and unrealistic result of earlier joints moving much more than later links in the linkage.

Differential-based techniques utilise an iterative approach that requires multiple step to reach a solution, such as those by ... Most inverse kinematics systems tend to use the Jacobian inverse method, despite the fact that this method suffers from singularity problems. The reason these techniques remain in widespread use is due to their relative ease of implementation and speed for real-time applications. Since all joint angles are updated in a single step, the linkage movements are dissipated over all the links, which results in a more realistic and pleasing movement.

3 Procedure

In this section we discuss our approach to solving the kinematics problem. We started with a simplified case, by considering the two-dimensional situation where a planar object with a predetermined number of links is reaching towards a point in the plane. Our first attempt at a solution was to use forward kinematics where at each time-step we calculated the subsequent position using the current state and the desired position of the end-point. Our discussion starts with our Matlab implementation as we wanted to ensure that the method works as desired before exporting it to the OpenGL framework. In the meantime, we started working on the basic structure in OpenGL, for instance we drew the necessary primitives, such as triangles, squares and lines in two-dimensions, and proceeded by including some appropriate data structures and some primitive user interface.

The next step was to apply inverse kinematics in the case of our two-dimensional problem. We chose to use the Inverse Jacobian technique and implemented the method with both full inverse of the Jacobian matrix as well as the pseudoinverse. The motion of the object was displayed graphically, and we altered the number of limbs in order to test the performance of our method. At this stage the method still had a number of shortcomings even though it was restricted to motion in a plane. Firstly, the motion was not stable when the object was reaching towards a point that was outside the circular region. Secondly, the method favoured the movement of certain limbs without us explicitly specifying the constraints. We also improved the

OpenGL user interface by adding mouse capture, and added chain class with bones and joints. In both Matlab and OpenGL the system is described using relative angles, i.e. starting at the origin, each subsequent angle is defined in the coordinate system of the previous one (picture???).

The main questions we faced at this point had to do with local behaviour of each joint and the numerical method used to solve the optimisation problem. The movement of the first few joints (those closest to the origin) appeared significantly more restricted than the movement of the end joints. To solve this issue, we normalise to get global behaviour of the whole system as opposed to local behaviour of each joint. In addition, the simulation naturally slows down as it gets very close to the destination due to the nature of the numerical method which tends to oscillate around the minimum point. This behaviour can be advantageous (especially in robotics) as it results in slow-down motion as it gets close to reaching the goal (e.g. grabbing an object or touching a surface) so as to avoid a severe collision.

We also note that the motion of the system is task-dependent, hence favouring the movement of a number of selected joints is reasonable. For example, if we assume that the object we are modelling is an arm, and the motion is defined as the arm reaching for a nearby object, it is logical to assume that the rotation of the elbow joint will be favoured against the rotation of the shoulder joint, etc. Therefore, we introduce a weight vector that is used to control the importance of the motion of each joint, and added constrains on the angles.

We continued to work on the graphical implementation and improved it in a number of ways. The simulation and rendering were separated into two different threads so that the speed of the simulation does not affect the speed of rendering. What is more, we started displaying the trace left by the tip of the object which makes it easier to track the motion of the object and adds visual appeal.

At this stage the Matlab and the OpenGL implementations were still independent, so we started importing the numerical method from Matlab to OpenGL. The resulting model was two-dimensional, used inverse kinematics, could be adapted to any number of limbs/joints, and had a simple user interface.

Naturally, the next step was to transform the model to the three-dimensional space. We first adapted the graphical framework to three dimensions, i.e. we could rotate the camera, and place a target anywhere in the space, however the object and the motion were still confined to a plane.

Let us now consider the implementation in both Matlab and OpenGL in more detail. The following two sections give a detailed account of the issues encountered during the implementation process, the solution methods we employed and the explanation of why we chose to use the particular approaches.

3.1 Matlab

We began by implementing a simplified two-dimensional linkage system in Matlab, as shown in 3.1. This enabled us to simulate the motion of the linkage and experiment with different techniques before porting the final version of the code to OpenGL.

We define the degrees of freedom of the system by a vector of angles $\boldsymbol{\theta} = [\theta_1, \theta_2 ... \theta_N]^T$, where each element in the vector is the relative angle in degrees from one joint to the next, for N joints. All angles are measured positively in the anticlockwise direction and the first angle in the vector θ_1 is measured from the positive horizontal axis. The end-effector position in 3D

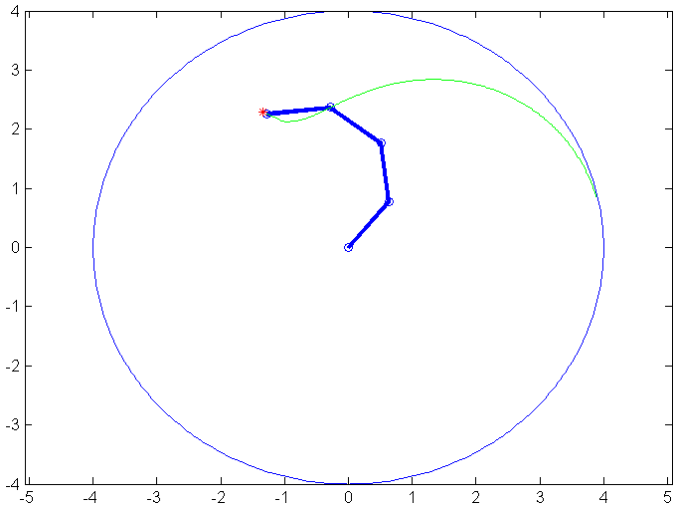

Figure 3.1: Two-dimensional Matlab implementation.

space is denoted by the vector $\mathbf{E} = [E_x, E_y, E_z]^T$. Using forward kinematics, we can compute the end-effector position from the given joint angles:

$$\mathbf{E} = f(\boldsymbol{\theta}) \tag{3.1}$$

The goal of the inverse kinematics problem is to compute the vector of joint angles $\boldsymbol{\theta}$ given a desired end-effector position

$$\boldsymbol{\theta} = f^{-1}(\mathbf{E}) \tag{3.2}$$

However this is difficult to solve because the equations are usually nonlinear and multiple possible solutions can often exist, or sometimes maybe no solutions. It is also challenging to solve analytically for more complex linkages with large numbers of links, so to solve this iteratively, we can employ the Jacobian method. First we define the Jacobian matrix as the matrix of partial derivatives

$$J = \begin{bmatrix} \frac{\partial f_x}{\partial \theta_1} & \frac{\partial f_x}{\partial \theta_2} & \cdots & \frac{\partial f_x}{\partial \theta_N} \\ \frac{\partial f_y}{\partial \theta_1} & \frac{\partial f_y}{\partial \theta_2} & \cdots & \frac{\partial f_y}{\partial y} \\ \frac{\partial f_z}{\partial \theta_1} & \frac{\partial f_z}{\partial \theta_2} & \cdots & \frac{\partial f_z}{\partial \theta_N} \end{bmatrix} \tag{3.3}$$

Where the Jacobian relates a small change in angle to a small change in the position of the end-effector

$$\partial \mathbf{E} \approx J(\boldsymbol{\theta}) \partial \boldsymbol{\theta} \tag{3.4}$$

There are a number of ways to compute the Jacobian, and one common way of evaluating it is to use the cross product of the axis of rotation (which for a two-dimensional system is simply the z-axis out of screen) with the vector from the joint position to the target end-effector position [1]. Therefore each column in the Jacobian, corresponding to the ith joint in the linkage, is given by

$$\frac{\partial \mathbf{E}}{\partial \theta_i} = \mathbf{v}_i \times (\mathbf{E} - \mathbf{P}_i) \tag{3.5}$$

Where $\mathbf{v}_i$ is the axis of rotation for the ith joint, and $\mathbf{P}_i$ is the position of the joint. This is implemented in the function calcJacobian.m. However, in our final implementation, we opt for a simple central difference approximation to the partial derivatives, which we found actually worked much better because it offered a more stable solution. This is implemented in the function fdJacobian.m

$$\frac{\partial f}{\partial \theta_i} \approx \frac{f(\theta_i + \delta) - f(\theta_i - \delta)}{2\delta} \tag{3.6}$$

Inverting the Jacobian matrix is an expensive operation. Since the matrix is often not square, the pseudo inverse can be implemented

$$J^+ = (J^T J)^{-1} J^T \tag{3.7}$$

However, another approach is to simply take the transpose of the Jacobian matrix. Although this method is less accurate than taking the inverse, the main great advantage of this method is that the problem of matrix singularities is avoided, since no matrix inversion is required. Taking the transpose is also much faster to compute than taking the inverse, so it is also advantageous for real-time interactive applications such as this. Once we have the inverse or transpose of the Jacobian, we can compute the required change in the angles

$$\partial \boldsymbol{\theta} \approx J^{-1} \partial \mathbf{E} \tag{3.8}$$

And then we perform a simple Euler integration to obtain the joint angles at the next time step Δt

$$\boldsymbol{\theta}^{t+\Delta t} = \boldsymbol{\theta}^t + \Delta t \partial \boldsymbol{\theta} \tag{3.9}$$

We must ensure the time step is sufficiently small to avoid numerical integration errors. The Euler method is also not the most stable of techniques, so perhaps Verlet or Runge-Kutta integration methods would work better. The complete Jacobian inverse/transpose algorithm implemented in Matlab is summarised below:

---

1. Calculate the difference between the goal position and the position of the end-effector

$$\Delta \mathbf{E} = \mathbf{G} - \mathbf{E} \tag{3.10}$$

2. Compute the Jacobian matrix J using the current joint angles and the appropriate method
3. Calculate the pseudo-inverse of the Jacobian J^+, or alternatively the transpose J^T
4. Determine $error = ||(I - JJ^{-1})\Delta \mathbf{E}||$
5. If $error > \epsilon$ then

$$\Delta \mathbf{E} = \Delta \mathbf{E}/2 \tag{3.11}$$

   and repeat step 4

6. Update the joint angles at each time step and use these as the new current values

$$\boldsymbol{\theta} := \boldsymbol{\theta} + \Delta t J^{-1} \Delta \mathbf{E} \tag{3.12}$$

   A simple way to add constraints and to prevent unnatural poses being computed is to cap the joint angles to upper or lower bounds. This requires the following additional step [6]:

7.

$$\boldsymbol{\theta} = \begin{cases} lowerbound & \text{if } x < lowerbound \\ upperbound & \text{if } x > upperbound \\ \boldsymbol{\theta} + J^{-1}\Delta \mathbf{E} & \text{otherwise} \end{cases} \tag{3.13}$$

3.1.1 Scaling considerations

In practice, the simple algorithm above does not always produce satisfying results. Even if the Jacobian, or angle steps are normalised, we can observe unnatural behaviour in the movement of the linkage. For example certain joints seem to move more freely than others, or at different times to others. Welman [7] suggests the following modification. We can attempt to compensate for this unnatural behaviour by introducing a scaling matrix $\mathbf{K}$, which is a constant diagonal matrix, whose ith diagonal entry $\mathbf{K}_i$ acts as a weighting factor for the computed joint velocity $\mathbf{q}_i$. The following term is suggested:

$$\mathbf{K}_i = \frac{1}{w_i \alpha_i} \tag{3.14}$$

where w_i is proportional to the length of link i, and is intended to offset the effects of the overall scale of the world. The α_i is a weighting factor for joint i, which can be thought as the responsiveness of the joint to an applied force. In future implementations, this parameter could be made available to the user to allow modification of the a joint's perceived stiffness.

Although Welman's proposed scaling matrix is currently a constant gain matrix, it would be relatively easy to make this into a time-varying matrix $\mathbf{K}(t)$. A very basic implementation could be to simply vary the ith scaling parameters in the matrix when the ith joint gets within a certain threshold distance from the goal position.

3.1.2 Incorporating constraints

Implementing constraints in an inverse kinematics system is very important if we want to provide some context to the system, rather than just dealing with an abstract robotic arm. For example, for a human skeleton, we would want to constrain the amount of rotation for specific joints, such as an elbow joint which doesn't have full flexibility of movement.

The current method of limiting the allowable joint angles simply by capping them when the angle update $\boldsymbol{\theta}^{t+\Delta t}$ exceeds the upper and lower bounds works reasonably well in practice, but is perhaps not the most elegant solution. It successfully prevents the joint angles from exceeding their limits, but it does not discourage the joints from reaching these limits. Another way to implement constraints would be to use a variant of the penalty method, which simply applies a restoring when the joint constraints are violated, to push the system back into a legal state in which all constraints are satisfied. This method was briefly explored in the Matlab implementation and implemented in the function evalF_constraints.m. A penalty term for each joint constraint is simply added to the cost function, which is the distance between the end-effector and the target position.

Perhaps the best way to incorporate constraints would be to use the Lagrange multiplier method, as we are essentially solving a nonlinear optimisation problem. This could be an interesting avenue in future work. In the final implementation of our system, we decided to stick with the simple constraint method explained in equation 3.13.

3.2 OpenGL

Our software was built on top of OpenGL version 4.3 so we can adopt the shaders introduced after 3.3 version, which uses a flexible pipeline. The following libraries are employed:

- GLFW for window managing,
- GLM for including the matrix operations and matrix-vector data types,

Listing 1: Renderer MVP calculation

```
viewProjection = projection * view;
//Calculate Model View Projection matrix
glm::mat4 MVP = viewProjection * modelMat;
```

- Boost provides smart pointers (which simplify memory management),
- GLEW which efficiently check which OpenGL extensions are supported on the platform that is used.

The C++11 standard is used as it introduces better default thread and mutex libraries as well as other improvements.

First, we consider all the necessary conversions that let us display the three-dimensional world on a two-dimensional screen surface. Initially we have the z value set to zero, and hence worked in a two-dimensional space. In our world model each object has it's own model transformation matrix M. We also have a camera matrix that defines the camera position and orientation. Lastly, a projection matrix describes the mapping of a pinhole camera which plots the three-dimensional world onto a two-dimensional screen. For every object in the scene, pixel/screen coordinates $[u, v, 1]$ are defined by multiplying the vertices of each object by the Model-View-Projection (MVP) transformation matrix which gives the position of each pixel on the screen, i.e. transformed vertex = M $\times$ V $\times$ P $\times$ object vertex.

In the first version of our software, we would create all the OpenGL buffers for each object on each frame and destroy them at the end of each frame, which was inefficient. The problem was fixed by preallocating buffer objects (that store unformatted memory on the graphics processing unit (GPU)) for each object at the start of the simulation, and then binding the framework to allocated storage (setting as the active), rendering and unbinding.

The initial implementation consisted of lines, squares and triangles, and an orthographic projection was used to represent the objects in two dimensions. At this point we added the chain of rectangles and lines where lines represented bones and rectangles were joints. A 4×4 rotation and translation matrices were introduced, so for each joint and at each time step the following calculation is performed: $\prod_{i=0}^{I} R_i T_i$ where R_i stands for the rotation of the joint and T_i corresponds to the translation of each bone.

In order to achieve a fluid and efficient rendering, we made the simulation process independent of rendering. Simulation and rendering are performed on two different threads, and have to access a shared resource concurrently. The synchronisation of these two processes is encapsulated in simulation controller class which uses a mutual exclusion lock variable. The rendering process has access to all the chain data while the simulation only uses the angles and lengths (taken from the copy of the chain) to perform the numerical calculations which advance the motion in time. The simulation runs almost continuously, it stops only to allow the renderer to look up the angle values each time it finishes rendering. This synchronisation mechanism controls the access to the data shared between the simulation and rendering threads.

For visual appeal, we draw a trail that follows the chain tip. A circular array is used for efficient rendering; a fixed buffer is allocated on the GPU at the beginning of the simulation and we are tracking the movement of the tip of the chain. Each time the tip moves a sufficient (predetermined) amount, we record the current position and include it in the circular array. Once we run out of space, we return to the start of the array and replace the first value with the

Listing 2: Common draw call for all VertexObjects.

```
void VertexObject::draw(Renderer& renderer) const {
        // Bind VAO and Buffers as the active ones.
        glBindVertexArray(vao);

        //Enable vertex buffer
        glEnableVertexAttribArray(0);
        glBindBuffer( GL_ARRAY_BUFFER, vbo);
        glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 0, (void*) 0);

        //Enable the other buffers
        <...>

        //Draw count objects of mode type
        glDrawElements(primitivePar.getMode(), primitivePar.getCount(),
        GL_UNSIGNED_INT, (void*) 0);

}
```

new one. Then the rendering is performed sequentially with all remaining values unchanged until we again reach the end of the array. This allows us to make only small uploads of data to he GPU.

At this stage we added the simulation code originally developed in Matlab to our the two-dimensional model, see Figure 3.2. Instead of calculating the pseudo-inverse and the error, only the transpose of the Jacobian is calculated. To allow user input, we draw a triangle which could be moved using the mouse, and was used to mark the goal position. The mouse position was transformed using an inverse orthogonal projection.

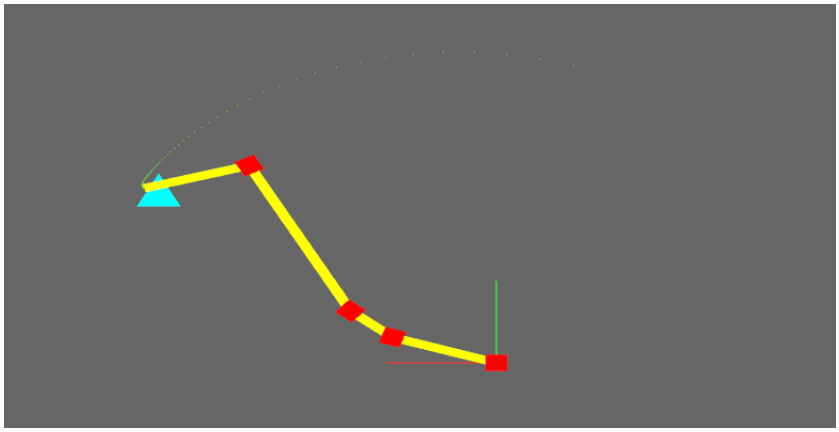

Figure 3.2: Final two dimensional simulation with the trace and a triangle showing the goal position.

One of the problems we faced had to do with the simulation thread accessing the rendering state, causing the software to fail if the frame rate was too high. The problem was solved by allowing the simulation to access only the simulation data. Here we also used data encapsulation; a new class, called ChainModel, was created that only had the simulation data. The Chain class is a subclass of the ChainModel and with added attributes and methods for rendering purposes. The same applies to bone and joint classes.

Listing 3: Simulation controller thread synchronisation.

```
void SimulationController::executeSimulationLoop() {
        while (simulating) {
                lock.lock();
                simSolver.solveForStep(goal, stepSize);
                lock.unlock();
                std::this_thread::sleep_for(sleepTime);
        }
}

void SimulationController::updateChain() {
        lock.lock();
        simSolver.updateChain();
        lock.unlock();
}
```

The simulation thread is either simulating or waiting for a new goal. It also checks when it's close to the goal and it stops at a given threshold, so the loop that updates the Jacobian matrix is not called though the simulation thread is still running. The user can input a new goal position at any time, as the simulation loop will then solve for the new goal using the current chain state.

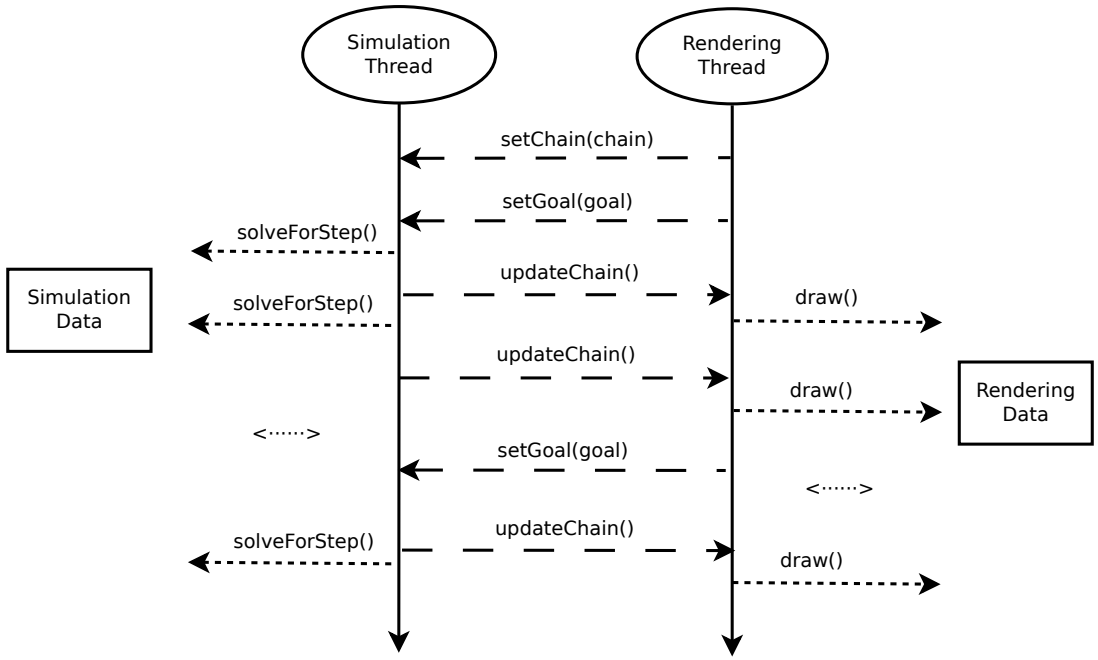

Figure 3.3: Diagram showing the data communication among the simulation and the rendering thread.

Our final improvement involved adapting the simulation and the display to work in a three-dimensional case, see Figure 1.1. We construct the chain using rectangular parallelepipeds, while the shaders implemented a Gouraud shading algorithm.

At this stage the model was displayed in three dimensions but was still confined to moving in the plane. We added a second rotation angle resulting in a rotation with respect to the z-axis followed by a rotation with respect to the y-axis, i.e. $\mathbf{R}_z \times \mathbf{R}_y \times \mathbf{T}$. The resulting solver is fully three-dimensional.

Listing 4: Update total transformation with current joint rotation and translation.

```
currentMat = currentMat * glm::rotate(joints[i].getZRotAngle(), zAxis)
                        * glm::rotate(joints[i].getYRotAngle(), yAxis)
                        * glm::translate(glm::vec3(bones[i].getLength(), 0, 0));
```

One of the objectives was to have controllable slow in and slow out motion at the start and end of each simulation. This was achieved by normaloising the total change in the change of the tip position. The change was scaled linearly, see Figure 3.4.

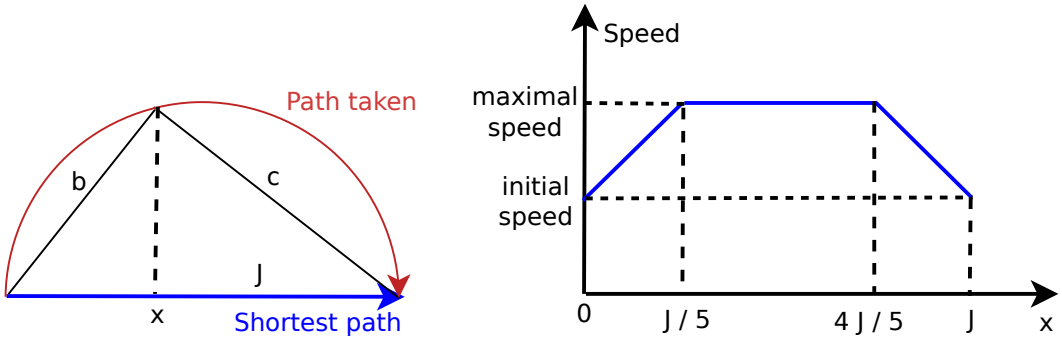

Figure 3.4: Diagram illustrating the linear scaling that results in slow in and slow out motion of the chain.

4 Conclusion

We have used the method of inverse kinematics to create a three-dimensional linkage that moves towards a specified goal position. The location of the goal in three-dimensional space can be adjusted by the user through keyboard input, and it's position can be changed mid-way through the simulation so that the linkage reacts accordingly. Although the rendering and lighting of the linkage is relatively simplistic in it's current implementation, it does effectively display the three-dimensional nature of the dynamic model. The inclusion of the tracer that describes the path of the end-effector is an effective addition, which emphasises the movement of the linkage and provides a visually pleasing result.

A number of parameters have been included that enable the user to control the physical properties of the linkage. The parameters must be hard set within the source code, prior to the simulation, but ideally we would prefer to set them in run-time through the use of a gui. For example, the rate at which the joints move can be altered so that certain joints appear to be more flexible than others. Constraints on the joints can be included by specifying the maximum and minimum allowable angles for the whole chain. In addition, the speed of the linkage can be tuned to provide a more organic behaviour; so that the linkage accelerates at the start of its motion and decelerates towards the end.

We provide a real-time OpenGL implementation of the inverse kinematics problem for an arbitrary number of joints, which runs smoothly, and solves for any target position within the range of the linkage. However, the system can suffer from oscillations when the target point is just inside the boundary. The system is also slower to solve when the target point is placed in

certain positions, for example if it is placed at a location incident with the chain when the chain is fully stretched.

4.1 Future Work

A number of directions remain for future work. We would like to experiment with different shading models to achieve higher realism. Another area of future work is to include a graphical user interface to give the user direct control of the simulation parameters, such as the stiffness, and the slow in and slow out motion. It would be interesting to explore other non-linear optimisation methods, such as Newton's method which incorporates the higher order derivatives. In addition, alternative ways of implementing the constraints could be tested, for instance the method of Lagrange multipliers. Finally, we are interested in loading three-dimensional mesh objects and textures to replace the geometric shapes used in the current version. The number of end effectors could be increased to give the object more character and charm.

References

[1] S.R. Buss. *Introduction to Inverse Kinematics with Jacobian Transpose, Pseudoinverse and Damped Least Squares methods*, 2009 (accessed October, 2014).

[2] K.W. Chin. Closed-form and generalized inverse kinematic solutions for animating the human articulated structure, 1996.

[3] J. Craig. *Introduction to Robotics: Mechanics and Control.* Addison-Wesley, 1955.

[4] M. Fedor. Application of inverse kinematics for skeleton manipulation in real-time. In *Proceeding of Spring Conference on Computer Graphics 2003*, pages 201–210. Slovak University of Technology in Bratislava, 2003.

[5] C. Kwang-Jin and K. Hyeong-Seok. On-line motion retargetting. *The Journal of Visualization and Computer Animation*, pages 223–235, 2000.

[6] M. Meredith and S. C. Maddock. Real-time inverse kinematics: The return of the jacobian. *Department of Computer Science Research Memorandum CS-04-06*, 2004.

[7] C. Welman. Inverse kinematics and geometric constraints for articulated figure manipulation. Master's thesis, Simon Fraser University, September 1993.