Department Of Computer Science, University of Bath

Inverse Kinematics Coursework

CM50244 Computer animation and games I

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

November 19, 2014

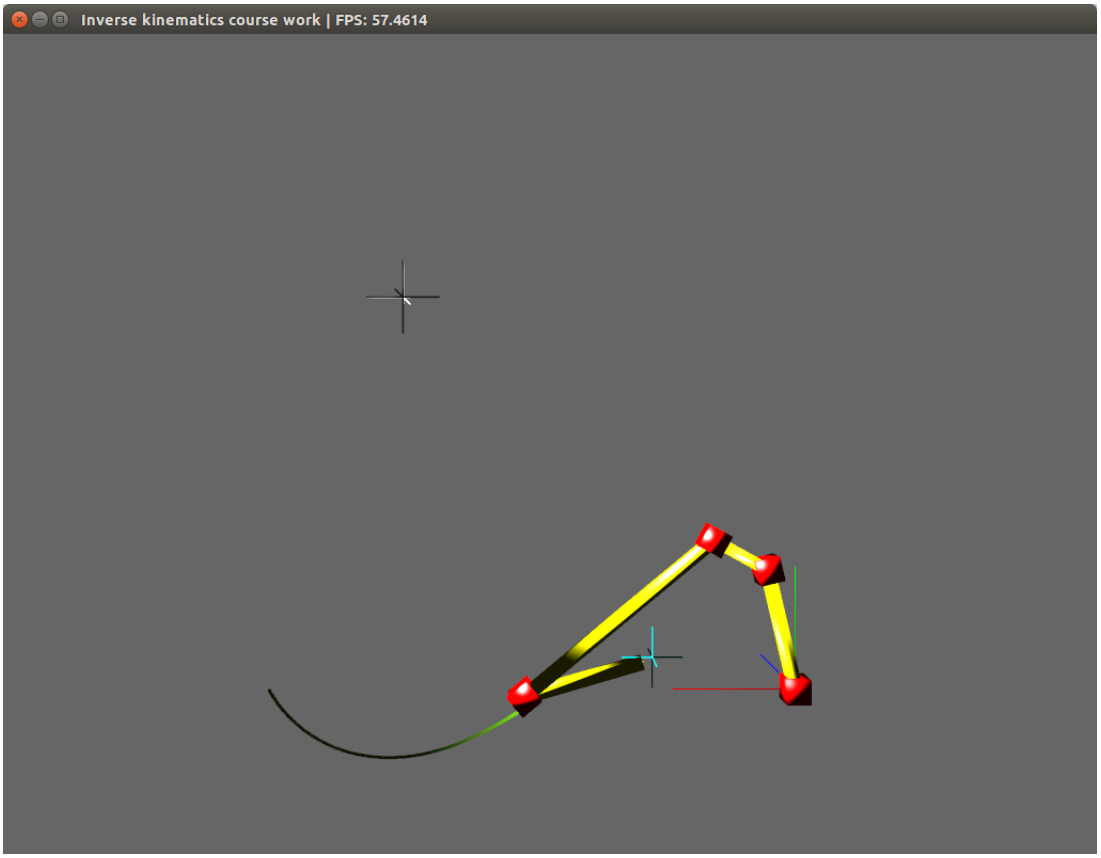

Figure 1.1: Final three dimensional simulation with the trace and a star showing the goal position.

1 Introduction and Objective

In this project we explore the application of inverse kinematics in three-dimensional space. First, we construct a simple composite object which contains long rigid parts that are connected so as to allow flexibility of movement. The joints connecting the rigid rods are restricted and can either rotate in plane and act like simple hinges. Second, the method of inverse kinematics is explored in order to determine the configuration of the object, i.e. the relevant angles of the joints, when the object is given a task. The method works by planning the motion towards a goal, which is a point in three-dimensional space subject to constraints. Finally, the movement of the object is animated, and a number of non-physical (?) parameters are adjusted to deliver a visually appealing performance.

This report provides a brief account of the objectives of the project and the methods that were explored. Moreover, the issues that were encountered are discussed including problems concerning both the computations and the implementation. We discuss the implementation in Matlab and the computational method that we use for inverse kinematics. We proceed by explaining the operations and execution in OpenGL. The last section of this report includes a review of the performance and the limitations of the method, as well as a discussion of possible further improvements.

2 Related Work

There are a number of well-established methods of approaching the inverse kinematics problem. Most inverse kinematics systems tend to use the Jacobian inverse/transpose method, despite the fact that these methods suffer 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.

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. Although this method is relatively simple and fast, it suffers from inaccuracies.

3 Procedure

In this section we discuss our approach to solving the problem at hand. We started by considering the corresponding 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. The first implementation was done in Matlab 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 rendering does not affect the speed of the simulation. 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 Start with Matlab

1. Issues
2. How we solve them?
3. Why we use what we use? (e.g. Inverse Jacobian)

We define the degrees of freedom of the system by a vector of angles $\boldsymbol{\theta} = [\theta_0, \theta_1 ... \theta_N]^T$, 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_0 is measured from the positive horizontal axis. The end effector position in 3D 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

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

To solve this using the Jacobian method, first we define the Jacobian matrix as the matrix of partial derivatives

$$J = \begin{bmatrix} \frac{\partial E_1}{\partial x} & \frac{\partial E_2}{\partial x} & \dots & \frac{\partial E_N}{\partial x} \\ \frac{\partial E_1}{\partial y} & \frac{\partial E_2}{\partial y} & \dots & \frac{\partial E_N}{\partial y} \\ \frac{\partial E_1}{\partial z} & \frac{\partial E_2}{\partial z} & \dots & \frac{\partial E_N}{\partial z} \end{bmatrix} \tag{3.3}$$

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 2D system is simply the z axis out of screen) with the vector of the joint to goal position. However we opt for a simple central difference approximation of the partial derivatives, which we found actually worked much better.

$$\frac{\partial f}{\partial x} \approx \frac{f(x+\delta) - f(x-\delta)}{2\delta} \tag{3.4}$$

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

$$J^+ = (J^T J)^{-1} J^T \tag{3.5}$$

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

$$\partial \boldsymbol{\theta} = J^{-1} \partial E \tag{3.6}$$

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

$$\boldsymbol{\theta}^{t+\Delta t} = \boldsymbol{\theta}^t + \Delta t \partial \boldsymbol{\theta} \tag{3.7}$$

We must ensure the time step is sufficiently small to avoid numerical integration errors.

3.2 Port the code to C++, 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, 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 = MVP * object vertex.

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

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 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.

Next, we set up the necessary objects for rendering. We create a vertex object class that encapsulated the buffer objects used by OpenGL. The vertex specification is performed using the Vertex Array Object (VAO), and each vertex object has its own VAO attribute. For the two-dimensional model we adopt a simple shader program without lighting and we specify the colour directly. The actual implementation uses a vertex buffer object (VBO) for vertices so that each object has only the vertices listed sequentially in the corresponding buffer, and we have a colour buffer (CBO) of an identical size that contains the corresponding colours. The element buffer object (EBO) contains indices to a given vertex; it allows us to avoid replication of vertices and draw more efficiently. In particular, to define any rectangle (quad) that is a combination of two triangles, we only need to specify four vertices instead of using all six.

The initial implementation consisted of lines, squares and triangles, and an orthographic projection was used to represent the objects in two dimensions. To gain insight into the performance of our model, we included a frames per second counter which indicates how fast the scene is being rendered. In order to draw an objects, we implemented a `Drawable` class. All the objects that need to be displayed have to implement an interface defined by the `drawable` class and possess a function called `draw`. The window draws drawable objects and does not distinguish between them. The line, square and triangle are all subclass of the VertexObject class which itself is a subclass of Drawable. 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 its total transformation matrix M is calculated as follows: $M = \prod_{i=0}^{I} R_i T_i$ where R_i stands for the rotation of the joint and T_i corresponds to the translation of the bone.

```
class VertexObject: public Drawable {
        <...>
protected:
        //Model matrix
        glm::mat4 modelMat;
        //Model vertex related data
        std::vector<glm::vec3> vertices, normals, colors;
        std::vector<unsigned int> indices;
        glm::vec3 centroid;

        //Common object and buffers needed for OpenGL rendering
```

```
        GLuint vao, vbo, nbo, ebo, cbo;
};
```

```
//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);

}
```

In order to achieve a fluent 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 allows the simulation and the rendering to perform their respective tasks as quickly as possible without any data corruption.

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 new one. Then the rendering is performed sequentially with all remaining values unchanged until we again reach the end of the array. This let's us 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. 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.

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 high (e.g. on the NVIDIA card). 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.

```
//Simulation controller thread synchronization
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 in a loop 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 update Jacobian loop 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.

Our final improvement involved adapting the simulation and the display to work in a three-dimensional case. We construct the chain using rectangular parallelepipeds, while the shaders implemented a Gouraud shading algorithm. First, we added normal vectors to each vertex in every object; they point outwards from every vertices that is on the surface of the object, and for the vertices which are on the edges we use the mean of the neighbouring planar normals. Local illumination model combines the ambient, diffuse and specular light to give the colour intensities of the reflections, and the final shading is achieved by interpolating the vertices and normals.

In addition, we included keyboard input control. In the final version the user is able to control the camera angle by moving the mouse, and change the distance from the camera to the scene using the keyboard. In particular, we use the mouse to change the angle of the camera with respect to the `lookAtVector` and use the keyboard to move the camera position in a direction related to the `lookAtVector`, forward, backwards or side movements. The goal is moved using the camera position update (while right-clicking), where we update the position in the direction of the camera `lookAtVector`.

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. $\mathbf{R}_z\mathbf{R}_y\mathbf{T}$. The resulting solver is fully three-dimensional.

```
//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));
```

Final, a maximum number of simulation steps was specified to ensure that the simulation was not wasting CPU in case the goal was unreachable. Also, while the model is waiting for

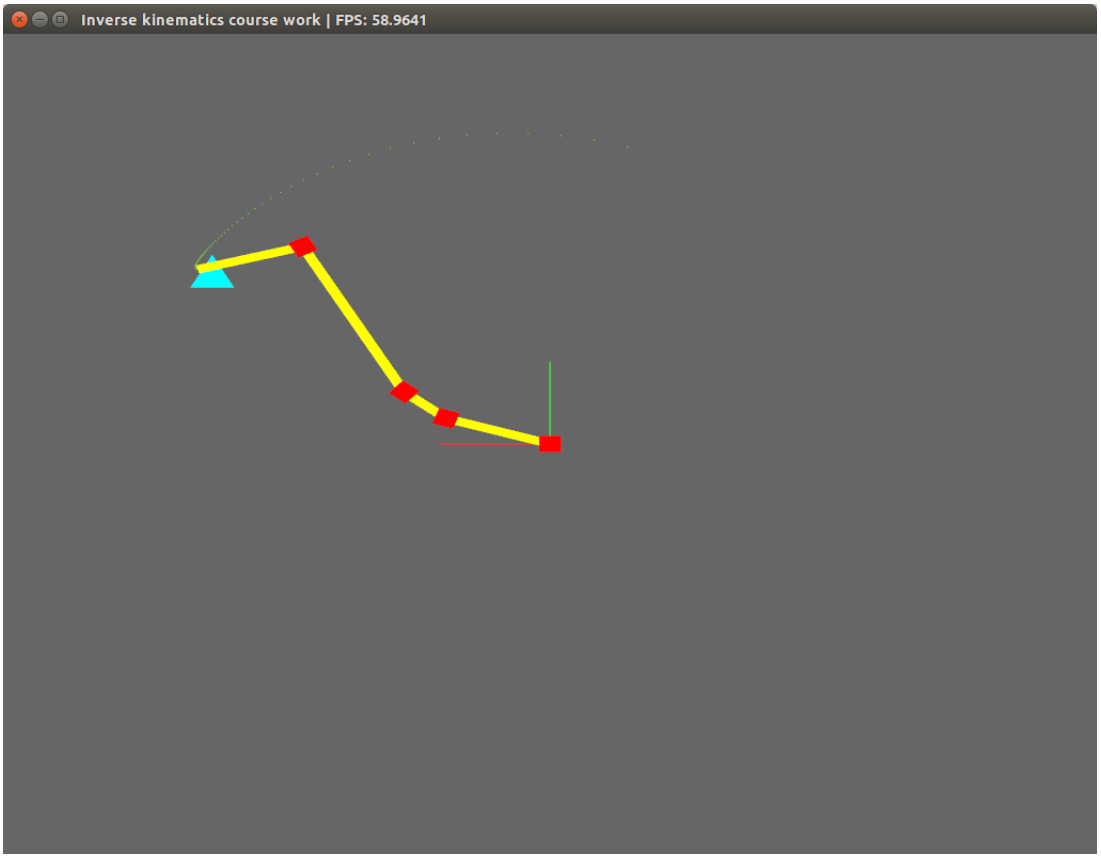

Figure 3.1: Final two dimensional simulation with the trace and a triangle showing the goal position.

a new goal, the simulation thread is stopped, and once a point goal is added, it goes back to simulating. We had to artificially increase the time in the simulation (now it performs a simulation step and then waits for 20 ms), because otherwise the time step had to be decreased to 10^{-6} which resulted in order to achieved a normal motion. Sleep time makes the computational load smaller on the CPU (i.e. we achieve the same accuracy with a lesser load on the CPU).

3.3 Exploration

The following properties are to be explored.

1. Some way of indicating where the remote end of the linkage should move to in 3-space. We have explored this already, see procedures section.

2. A way to change the physical properties such as rate at which the joints can change and the slow-out and slow-in of the movement of the end of the linkage as it leaves it current position and approaches the target position respectively.
We explored the possibilities in matlab, and, if time permits, we would implement it in c++.

3. Whether to model and thereby vary the flexibility of the rods. We explored the possibilities in matlab, and, if time permits, we would implement it in c++.

3.4 Summary and Conclusion

1. Give a short summary of the project.
2. What the final thing would look like if we had more time to work on it?
    - Load 3D mesh objects and textures instead of parallelepiped...
    - Add a better shading model
    - Add a GUI

References

[1] Martin 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.

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