Advanced Computer Graphics

Ikarus – Inverse Kinematics

John Bartholomew

jb5950

# Introduction

I chose to write an inverse kinematics solver. I have used the Cyclic Coordinate Descent algorithm, and have implemented root-changing and a basic joint constraint system.

[*I have had quite a lot of discussion with Mark Castell, who also implemented an inverse kinematics solver, however our implementations and reports are entirely separate.*]

# Algorithm

## Basic IK

I implemented a Cyclic Coordinate Descent (CCD) based IK solver. The basic CCD algorithm is described in [Bax00] and [Dra08]. CCD works by iterating over each joint along the chain from the end effector to the root of the hierarchy, and applying a rotation at that joint to minimize the distance between the end effector and the target. Visually, if you consider a vector from the joint being modified to the end effector, the rotation applied at the joint will make that vector point directly from the joint to the target. This process is repeated until the position of the end effector stabilises.

## Changing Root Node

Most inverse kinematics algorithms expect the armature to be in the form of a tree of bones – *i.e.*, they expect there to be some fixed ‘root’ bone. The pose of the armature is typically defined in terms of relative joint rotations so that rotating one bone will also rotate all of its descendant bones in the tree.

This means that the IK solver can’t change the position of the root bone in order to meet its target. Usually for animation purposes, the root of an armature for a human or animal character will be somewhere in the lower back, but this is often inconvenient for animation, where the position of the lower back need not be totally fixed and some other bone (for example, one of the feet) would make a more useful root.

From an algorithmic point of view, it usually doesn’t matter which bone is the root, as long as all of the joint rotations are defined relative to the correct ‘parent’ bone. So, I have implemented a system to allow you to select any bone as the current root.

The algorithm for changing the root is very simple:

1. A chain of bones is identified from the original root to the new root.
2. The rotation for each bone along this chain is set to be the inverse of the next bone’s rotation: this makes it a rotation relative to the next bone in the chain (the one closer to the new root) rather than being relative to the previous bone (closer to the original root).
3. The new root bone’s rotation is set to be its absolute orientation. This is calculated in the same loop by accumulating the original rotation values as we go.

## Constraints

I spent quite a long time thinking about how constraints for joints can be specified. Several options are considered in the analysis section, but I went with the following system:

Conceptually, an unconstrained joint provides a full 3 degrees of freedom (it allows any rotation). This rotation can be broken down into a direction vector, which will be somewhere on the unit sphere centred on the joint, and a ‘twist’ value which is a rotation around this direction vector. The direction vector can in turn be broken down into two angles, which I shall call elevation and azimuth.

Each joint has a direction which goes ‘out’ of the joint (*not* necessarily the direction of the bone in the neutral pose); the ‘elevation’ is given by the angle that the joint’s direction vector makes with this ‘out’ vector. The joint also has a ‘front’ and ‘side’ direction orthogonal to ‘out’, and one of these can be used to give the ‘azimuth’ angle.

In order to apply constraints, I first calculate the new rotation of the joint as in basic CCD. Then, the joint’s new rotation is broken down into elevation, azimuth and twist (this is done as in [Blo02]). Twist and azimuth are clamped into their constrained ranges. The constrained elevation is then calculated to minimize the angle between the constrained direction and the unconstrained target direction. This is not simply a clamping operation, because to minimize this angle the calculated elevation must be worked out in the context of the clamped azimuth.

This optimal elevation is calculated by projecting the target direction to be in the same plane as the azimuth line and then calculating what angle it is at on this plane. If this angle is within the range of valid elevations, it is used directly, otherwise the minimum and maximum elevations are checked to find which one is closer to the target direction, and that elevation is used.

# Results

What works well:

* The basic CCD algorithm is actually seems quite fast in simple situations.
* CCD seems reasonably robust.
* Changing the root is relatively simple and makes it trivial to, *e.g.*, ensure that a character’s foot stays in the same place while they reach for an object.
* Constraints seem to work fine for simple situations: when the joint is constrained to have two fixed angles (*e.g.*, twist and azimuth) and a range for the third angle; or when the joint is given an unconstrained azimuth but constrained elevation, etc.

What works badly:

* CCD is slow when the target is out of range and the joints must all ‘stretch out’ to get as close as possible.
* CCD does not generate ‘natural’ poses, and although it’s a relatively simple algorithm, I think it would be quite difficult to make it favour particular poses.
* The constrain system isn’t particularly flexible – it is sometimes difficult to define constraints for a joint which make sense and so it’s easy to end up with either over-constrained or under-constrained joints.
* Twist constraints are difficult to use correctly.

# Conclusions/Analysis

## Basic IK

The CCD algorithm can be broken down into the high-level algorithm –iterate over the joints in the kinematic chain and solve for each one individually – and the low-level calculations used to decide how to rotate an individual joint.

In Ikarus, the low-level calculation uses an analytic solution for the joint being updated, in order to minimize the distance from the end effector to the target. The joint is rotated such that the vector from joint-to-end-effector is in the same direction as the vector from joint-to-target-position.

Using this method means that CCD is actually very fast in situations where a joint can move the end effector significantly closer to the target, because it will rotate the joint by the full amount in a single step and so rapidly approach the target. This happens when the vectors from joint-to-target and joint-to-end-effector are pointing in significantly different directions.

However, there are disadvantages of rotating the joint with this mechanism:

* It’s difficult to apply joint constraints or to attempt to achieve secondary objectives in the pose, because the joint is rotating a large amount in a single step – this means that the constraint system must be able to work out the best constrained orientation for a joint given an initial orientation (from the basic CCD step) that is far outside the valid range, which can be quite difficult to calculate or even define.
* When constraints are applied, they can make the algorithm much slower to reach a solution. Because they’re applied independently of the CCD algorithm, it ends up working against the constraint system, with joints being repeatedly rotated a lot by CCD and then rotated back most of the way by the constraint system, to give a very ineffective movement overall.
* The resulting motion is not at all smooth, so it can’t be directly used for animation from one pose to another.
* The motion will always favour rotations of joints near the end effector over rotation of joints near the root, which is unrealistic. A cursory look at how people move indicates that which joint they use to move is dictated largely by the scale of movement being made – a large movement will be made by moving joints near the ‘root’ of the body (outwards from the chest and abdomen), with small adjustments made in joints near the end effectors.

## Root Changing

The root changing system worked quite well, although it does make the rest of the system slightly harder to write because nothing can rely on having a stable parent/child relationship between bones.

The main aim with the root changing system is to get around having a ‘special’ root bone (special because it can’t be moved by the IK system), but of course being able to change the root only half solves the problem.

I think it would be interesting to look at IK systems that don’t require a root bone at all – this would have to solve for multiple targets/effectors, since with no root any *single* target position can trivially be ‘solved’ for by a simple translation to put the effector in the right place. Having said that, translations themselves are a slightly subtle issue when there is no root bone – a pose doesn’t necessarily include location information, only the (relative) orientations of all the bones, which in turn defines the relative locations of each end effector.

## Constraints

There are two halves to the problem of implementing constraints. One is how to apply constraints or include them in the IK solver algorithm. The other is how to define the constraints in the first place.

The question of how to define joint constraints depends on how the joints are defined – one option (used in Ikarus) is that joints default to having a full three degrees of freedom (*i.e.*, allow any rotation). Another would be for each joint to only provide a single degree of freedom (rotation in a single axis), and to chain joints together with zero-length bones to achieve more degrees of freedom in the same place. I chose the first option partly because it makes a basic IK solver (without constraints) easier to make, and partly because the human body contains few truly one-degree-of-freedom joints, and I expected that composing joints would lead to problems with singularities making joints appear to ‘lock up’ – this intuition may have been flawed however.

Defining constraints on one degree-of-freedom joints should be relatively simple – it could be done by limiting rotation of the joint to a given range of angles. However, this would make it very difficult to simulate the kind of constraints that appear in a real human or animal. In a human the constraint on one degree of freedom of a joint is quite often dependent on the angle in another degree of freedom. For example, when you arm is in front of you, the shoulder joint allows it to point across your body, but if your arm is rotated to be pointing behind you, its rotation in the horizontal plane is much more restricted and you can’t make it point across your body (with the shoulder – of course there are extra degrees of freedom in the elbow that let you point your forearm across your body behind you).

If joints default to a full range of rotation then defining constraints on them is more difficult, but allows for more flexible and realistic ranges of motion. One way to define constraints on a joint which otherwise allows any rotation is to break the rotation into a direction vector and twist value. This direction can be limited to a region on the surface of a sphere centred on the joint, and the twist can be limited with a simple value range constraint.

There are several ways of defining the region on the sphere’s surface within which the direction vector is allowed to lie. In Ikarus, this region is defined with elevation and azimuth constraints, which is a fairly simple system but allows a reasonable approximation to many of the joints in the human body. If implemented naïvely by clamping the azimuth, elevation and twist values separately, it does have certain problems. This is what I initially implemented, but when doing so, the resulting direction is not necessarily ‘correct’ – it isn’t necessarily the closest direction vector that lies within the allowed region for the joint.

This is because the values of elevation and azimuth interact, so if the unconstrained direction vector is, for example, within the elevation band but outside the azimuthal band, clamping the azimuth value will put that vector onto the edge of the region, but at the same elevation, whereas if the unconstrained direction is on the opposite side of the sphere, then a better constrained direction would also be at one of the limits of elevation in order to minimize the angle between constrained and unconstrained direction vectors.

I improved this simple clamping by clamping azimuth and twist, but then calculating the optimal elevation value for the (clamped) azimuth, and if the resulting elevation is out of range, then checking both the minimum and maximum elevations and selecting the better one.

The constraint system could still be improved quite a lot however. It is currently difficult or impossible to define useful constraints for some joints. Simply putting limits on elevation, azimuth and twist is an inherently limited way of defining constraint shapes. Also, the constraints are defined in terms of the ‘orientation’ of the parent bone in the joint (this defines the ‘out’ vector for the joint and the direction of 0 azimuth) and this orientation is not controllable independently of the orientation of the parent bone itself, so if the ‘out’ vector is not what you want, there’s nothing you can do to fix it.

So, one way to improve the constraint system’s flexibility would be to let you explicitly define an orientation for the joint. A much more flexible system is described in [WG01], and is based on defining a polygon on the surface of the unit sphere around the joint, where the direction of the bone coming out of the joint is limited to lie within this polygon, and twist limits are defined at the vertices of the polygon and interpolated between them.

However, I think a much larger problem than the lack of flexibility in the constraint definitions is that the constraints are not integrated with the rest of the IK solver algorithm. There are better IK solver algorithms (for example, one based on linear programming, described in [HKL05]) which solve for joint angles with constraints taken into account throughout.

# References

**[Bax00] Bill Baxter, 2000**, University of North Carolina at Chapel Hill: Fast Numerical Methods for Inverse Kinematics - <http://billbaxter.com/courses/290/html/>

**[Blo02] Jonathan Blow, 2002**, The Inner Product articles: Inverse Kinematics with Quaternion Joint Limits - <http://number-none.com/product/IK%20with%20Quaternion%20Joint%20Limits/index.html>

**[Dra08] Spencer Drayton, 2008**, Spencer Drayton’s Blog: IK and Cyclic Coordinate Descent - <http://spencerdrayton.co.uk/blog/2008/10/15/ik-and-cyclic-coordinate-descent/>

**[HKL05] Ho, Komura and Lau, 2005**, Proceedings of the ACM symposium on Virtual Reality Software and Technology 2005, Pages 163-166: Computing Inverse Kinematics with Linear Programming - <http://doi.acm.org/10.1145/1101616.1101651>

**[WG01] Wilhelms and Van Gelder, 2001**, Journal of Graphics Tools, Volume 6, Number 2: Fast and Easy Reach-Cone Joint Limits - <http://jgt.akpeters.com/papers/WilhelmsVanGelder01/>