Inverse Kinematics On The Java 3D ä Scene Graph

Fred Klingener
klingener@BrockEng.com
Brock Engineering, Roxbury CT

Abstract

Kinematics, the study of the motion of bodies without regard to their masses or the forces causing their motion, has been around for centuries. The scene graph is the emergent standard hierarchcal data structure for computer modeling of three dimensional worlds, but kinematic models of machines or mechanisms that have external constraints or constraints that span interior nodes do not sit comfortably on its open-branched tree topology. Methods that have been developed to solve the kinematics of a constrained model tend to work backwards down the scene graph's branches from leaf to root, and the term "inverse kinematics" has been attached to them.

Inverse kinematics solvers have long been fixtures in computational modeling with particular activity in robotics, in human figure animation, and in proprietary CAD and machine design software. This paper describes a general approach to solving inverse kinematics on the Java 3D™ scene graph and illustrates the approach using a particular example from classical mechanics with a focus on methods suitable for high fidelity simulation of continuous processes.

Keywords: inverse kinematics, scene graph, Java 3D™

Categories and subject descriptors: I.6.8 [SIMULATION AND MODELING]: Types of Simulation---Continuous; J.2 PHYSICAL SCIENCES AND ENGINEERING---Engineering

1 INTRODUCTION

1.1 Kinematics

Kinematics studies the geometric properties of the motion of points without regard to their masses or to the forces acting on them. A set of points with the property that the distances between any two of them never varies is called a *rigid body* or *rigid link*. The position of a rigid body in space is defined by six dimensions, three translations and three rotations. A *kinematic chain* is a set of links connected by joints that constrain their relative movement. An *open chain* is a set of links (such as a common industrial robot) with one end attached to a rigid base . *A closed chain* may be attached to a rigid base in more than one place. A single loop closed chain attached at each end is commonly called a *mechanism*.

The *state* of a mechanism is the set of properties needed to completely determine the positions of all of its constituent parts. These properties are called *independent state variables*, and their count determines a mechanism's *degrees of freedom*. *Constraints* are limitations on the motion of a mechanism or of its constituent parts. A number of *dependent variables* may be required to determine the relative positions of a mechanism's parts, and they are derived from the *independent variables* and *the external constraints*. A *determinate* mechanism has the property that the number of dependent variables is equal to the number of external constraint conditions.

The links of a *planar mechanism* are constrained to move in a plane parallel to a base plane, usually by hinged joints whose axes lie perpendicular to the base plane. The axes of hinged joints of *a spherical mechanism* all intersect at a point. Joints of *spatial mechanisms* have no special relationship to a common point or plane. [1]

1.2 The Scene Graph

The *scene graph* is the emergent object-oriented hierarchical data structure for describing geometric relationships, appearance, and behavior in computer generated virtual reality worlds. It is an acyclic tree, meaning that each branch has a single attachment point, and therefore, it cannot form closed loops. The Java 3D API Specification [3] describes the construction of the scene graph tree, which uses BranchGroup nodes and TransformGroup nodes to connect the branches and uses Leaf Nodes to contain information such as geometric shapes, appearance, lighting, and behavior.

Throughout, this paper uses a particular example from classical mechanics to illustrate the general methods. The *four bar linkage*, a particular class of planar, determinate, single loop closed kinematic chains, has been the focus of study by mathematicians, philosophers, royalty and engineers for centuries, and, respecting tradition, this paper will employ it as the concrete example. The following describes how the scene graph can be used to describe its geometry and connectivity.

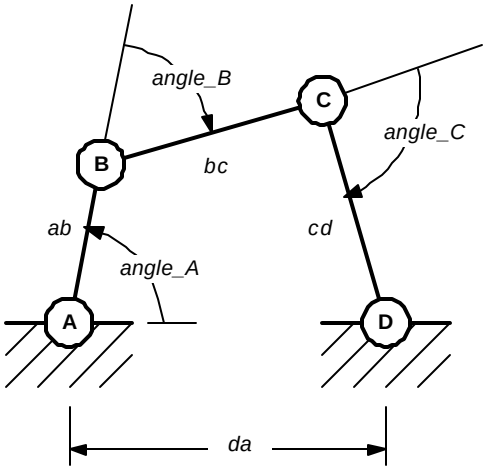

Figure 1. Four bar linkage.

Figure 1 shows a common configuration of a four bar linkage. Rigid links AB, BC, CD, DA are connected by hinged joints (A,

B, C, D); one of the links (DA) is considered to be fixed to a foundation, one link is considered to be a *driver* or *input link* (AB), the adjacent link is a *connector* or *drag link* (BC), and the last, the *driven link* or the *output*. (CD).

Figure 2 shows how a scene graph could be used to represent the geometric arrangement of the links of a four bar linkage. The Shape Leaf Nodes represent the geometries and visual properties of the individual links themselves, the Branch and Transform Group Nodes describe the structure of the tree that defines the geometric relationship between them, and the Behavior Leaf Node gives motivation to the input link AB.

The central failings of the bare scene graph are apparent from the figure. While the Behavior node can set the transformation for joint A by setting the independent variable `angle_A`, and the lengths of the rigid links `ab`, `bc`, and `cd` determine the translations, the rotations at joints B and C are unknown, and the requirement that the 'D' end of the link CD engage the joint D fixed on the base remains unexpressed on the tree's acyclic topology.

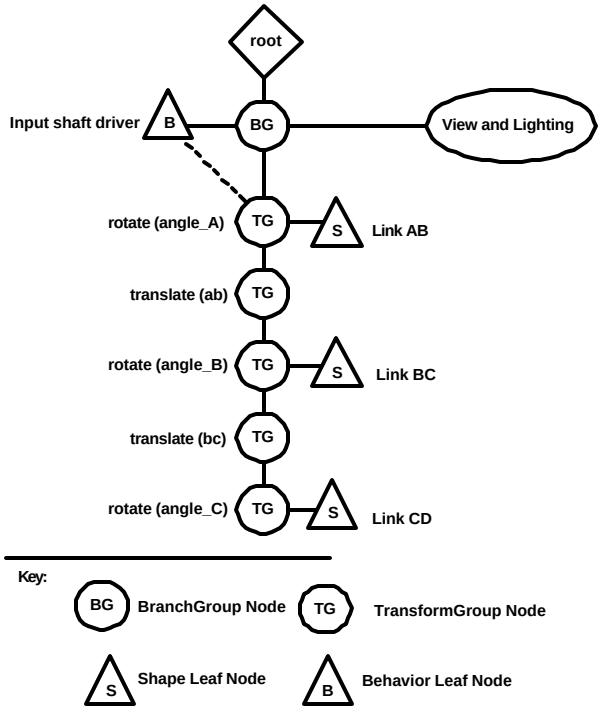

Figure 2. A four bar linkage on a Java 3D™ scene graph

2 APPROACH

This section describes a three-pronged approach to using inverse kinematics for solving external constraints on an animated mechanism model in Java 3D™. The first sub-section describes the mathematical background, the solution sequence, and the organization of the scene graph model, the second describes the approach to creating a Constraint class to organize and expedite geometric calculations in support of the solution, and the third sub-section describes the standard way that simple Java 3D™ models are animated, the inherent problems, and an approach to modifications necessary to permit stable and efficient computation.

2.1 Solving External Constraints

Typically, constraints can be expressed in a number of equations or inequalities that describe the relationship among machine parts or between machine parts and the foundation. This paper considers a subset of constraints that can be expressed in equations whose terms are time-independent. The example has constraints that set displacements of a mechanism part equal to zero relative to a fixed foundation point. Further, the determinate mechanism discussed in this paper has an equal number of dependent variables that can be set to satisfy those equations. The mathematical problem then reduces to solving N nonlinear equations in N unknowns.

$$F_i(x_1, x_2, x_3, \ldots x_N) = 0 \qquad i = 1\ldots N \tag{1}$$

where F is a function of x, the dependent variables. For very simple mechanisms, the equations can be solved from quadratic or trigonometric closed forms, but numerical marching techniques are preferable for mechanisms of any complexity.

Press et al. [2] describe the Newton-Raphson method, simplest of the numerical techniques for solving sets of simultaneous nonlinear equations. The method begins with an initial guess for the dependent variables (x_j), measures the resulting values of the functions F, calculates and applies a change to the dependent variables and repeats to convergence.

In the general case, the equation set (1) can be particularly troublesome when it represents kinematic constraints, because it may have many, one, or no solutions. According to Press, there are no good methods to find global solutions to the set, and accordingly, the methods described here are applicable only to mechanisms for which there are known solutions and only to stable regions for those mechanisms.

The key to stability and rapid convergence is an initial guess not too far from the final result. The changes required to the dependent variables are estimated by estimating the local derivatives of F_i with respect to x_j, then using them to compute the new x_j required to drive F_i to zero. The derivatives $\partial F_i/\partial x_j$ are measured by incrementing each x_j in turn and measuring the changes to F. The $N \times N$ matrix $\partial F_i/\partial x_j$ is called the Jacobian, J. In vector form:

$$\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = \mathbf{F}(\mathbf{x}) + \mathbf{J} \times \delta\mathbf{x} + (\text{higher order terms}) \tag{2}$$

Setting $\mathbf{F}(\mathbf{x} + \delta\mathbf{x}) = 0$ to satisfy the constraint equations, and neglecting higher order terms

$$\mathbf{J} \times \delta\mathbf{x} = -\mathbf{F}(\mathbf{x_{old}}),$$

which is a set of N linear equations in N unknowns. It may be solved for $\delta\mathbf{x}$ by one of several different methods, such as direct inversion of the Jacobian

$$\mathbf{x_{new}} = \mathbf{x_{old}} + \delta\mathbf{x}$$
$$\mathbf{x_{new}} = \mathbf{x_{old}} + \mathbf{J}^{-1} \times (-\mathbf{F}(\mathbf{x_{old}})), \tag{3}$$

and the steps are repeated until $\mathbf{F}(\mathbf{x_{old}})$ are sufficiently close to $\mathbf{0}$. For a high fidelity simulation, this procedure is executed and the dependent variables are set prior to the rendering of each frame.

2.2 The Constraint Class

The approach to designing the Constraint class is motivated by the need to close the loops left open on the scene graph and by the desire to organize and encapsulate the methods required to measure the constraint mismatches and the elements of the Jacobian described in sub-section 2.1 First, the Constraint class can be used to close a loop on the scene graph by extending (subclassing) Group so that it can itself be placed as a component on the scene graph (to fix the geometric frame of reference in which the constraint displacements are to be measured), and by containing, as a member, a reference to another scene graph Node (to identify the position of the constraint's mating part). Second, a Constraint

instance can exploit the computational power of Java 3D™ in the following way. Because a Constraint instance and its referenced Node each have particular positions in space by virtue of their positions on the scene graph, their relative positions (usually the mismatch between the current and desired displacements at the constraints) can be readily extracted by a few simple calls to Node geometry methods.

2.3 Java 3D™ Animations

The Java 3D™ API supports some types of simple animations. Interpolator objects, for instance, are subclasses of Behavior, so not only can they monitor the system clock and provide smooth and continuous modification of TransformGroups that set the independent variables of a scene, but they can also control other parts of the scene (such as computation of the dependent variables) by means of overrides of their `initialize()` and `processStimulus()` methods.

However their design details make the standard Interpolators unsuitable. Because their time base (and thus the motion they impart to their target scene objects) is rooted in the system clock, the interpolated motion they impart can proceed completely independently of the rendering. While the CPU is off resizing windows, for example, the independent input variables of the invisible mechanism are advancing, while the Interpolator's `processStimulus()` method remains uncalled.

Sub-section 2.1 described the usefulness of Newton-Raphson method of iteratively updating the dependent variables for each rendering frame by taking, as the initial guess, the values of the dependent variables from the immediately prior frame. But the stability and convergence of the method depended on the close proximity of the initial guess to the final answer. Clearly, large jumps must be avoided.

The solution is a new Behavior class that creates and maintains its own simulation clock that advances by increments in the `processStimulus()` method. In this way, the scene's independent variables do not advance unless the `processStimulus()` method is called, and as a result, the dependent variables from the last time step are appropriate as initial guesses for the Newton-Raphson method.

3 IMPLEMENTATION

Figure 3 shows a scene graph that represents the four bar linkage of Figure 2 with the changes required to control it and to resolve the constraints. Figure 3 uses solid lines to connect tree parts in the usual way. The dotted lines indicate functional interactions among parts, and the boxes with the grayed boundaries identify special classes along with the numbers of the sub-sections in which they are discussed here. The figure shows the spine of the linkage down the left hand side, with the behavior setting the independent variable *angle_A* as before. The kinematic constraint is reflected in the closed loop formed on the right hand side by a Constraint object, `PinD`, mounted on the scene graph and referencing the D end of the link CD.

The dotted lines identify interactions among the objects, along with an identification of the process step (described below) in which they are active. The solution of the inverse kinematics occurs before rendering each frame, and it proceeds with the following steps:

step 1.) the `MachineController` object (a Behavior) wakes up, updates its simulation clock, resets its wakeup alarm, reads its interpolator, sets the new value of the independent variable *angle_A*, and then in step 2.) calls the `FourBarLinkage` method `solve()`. In step 3.), that `solve()` method performs the calculations needed to solve the constraints. That step is divided into three sub-steps. In step 3a.), the `Constraint` object reports the mismatch at the constraint for the initial guess. In step 3b.) the `solve()` method increments the dependent transforms in turn, and the `Constraint` object reports the resulting displacement changes in step 3c.) The `solve()` method then composes the Jacobian, inverts it, multiplies by the negative of the original mismatch, and in step 4.), it sets the dependent variables to their fresh values.

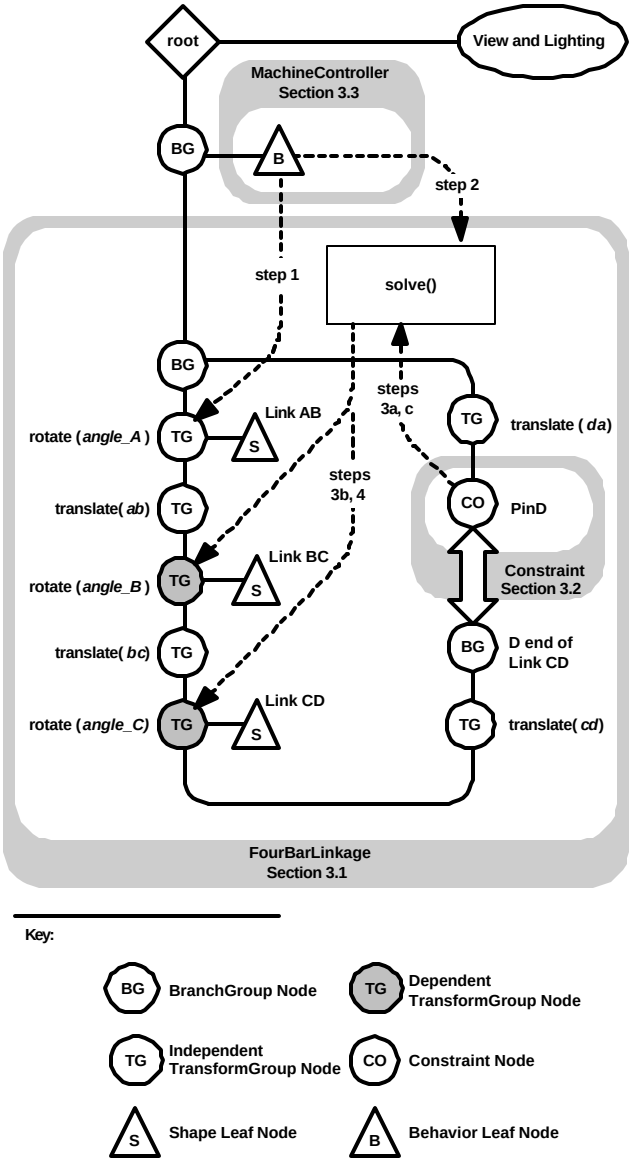

Figure 3. Scene Graph with Constraints

The following sub-sections describe the classes required - outlining their inheritance and the additional members and methods required. Section 3.1 describes the construction of the model itself and its `solve()` method, section 3.2 the `Constraint` class, and section 3.3 the Behavior that controls its operation.

3.1 The FourBarLinkage Class

Section 2.1 described the mathematical underpinnings of the approach to solution of the inverse kinematics constraint equations. This section describes the implementation details for the example

four bar linkage shown on Figure 1. The `FourBarLinkage` class extends BranchGroup, constructs the mechanism parts, sets the appropriate capability bits, and assembles them in the usual way. It has an `initialize()` method that can be called from the `MachineController`'s `initialize()` method. The meat of the class, though, is its `solve()` method. Called each frame by the `MachineController`'s `processStimulus()` method after that method has set the new independent state variables, `solve()` finds the roots of the constraint equations and sets the independent variables.

In our example linkage, when the `solve()`method receives the mechanism, the independent variable (*angle_A*) has been set to the next frame, the dependent variables (*angle_B* and *angle_C*) remain as they were for the last, and there are, in general, displacement mismatches, *mx* and *my* in the x and y directions at the constraint point D. Before the frame can be rendered properly, this displacement mismatch must be eliminated by finding new values for *angle_B* and *angle_C*, and the corresponding TransformGroups must be set. The condition of the mechanism, then, is ready for the Newton-Raphson method.

The four bar linkage has one independent variable, *angle_A*, two dependent variables, *angle_B* and *angle_C*, and two constraint conditions *Dx = 0* and *Dy = 0*, where *Dx* and *Dy* are the displacements in the x and y directions of the 'D' end of link CD with respect to the foundation point D. Equations (1) for the four bar linkage become:

$$Dx(angle\_B, angle\_C) = 0$$
$$Dy(angle\_B, angle\_C) = 0$$

Recalling that the Jacobian has one row for each constraint condition and one column for each dependent variable, the Jacobian for the example four bar linkage is a square 2×2 matrix with the form

$$J = \begin{vmatrix} \partial Dx / \partial(angle\_B) & \partial Dx / \partial(angle\_C) \\ \partial Dy / \partial(angle\_B) & \partial Dy / \partial(angle\_C) \end{vmatrix}$$

The elements may be approximated empirically on the scene graph itself by first incrementing *angle_B* by some known *D(angle_B)*, reading the resulting change in *Dx* and *Dy*, thus obtaining the first column of the Jacobian, and second by resetting *angle_B*, and incrementing *angle_C* by some *D(angle_C)* and measuring the resulting *DDxc* and *DDyc* to obtain the entries for the second column. The resulting approximate Jacobian has the form:

$$J \approx \begin{vmatrix} \Delta Dxb / \Delta(angle\_B) & \Delta Dxc / \Delta(angle\_C) \\ \Delta Dyb / \Delta(angle\_B) & \Delta Dyc / \Delta(angle\_C) \end{vmatrix}$$

For illustrative purposes, the required adjustments to the to the dependent variables *correction_B* and *correction_C* are solved by inverting the Jacobian and multiplying it by the negative of the original mismatch.

$$\begin{vmatrix} correction\_B \\ correction\_C \end{vmatrix} = J^{-1} \times \begin{vmatrix} -mx \\ -my \end{vmatrix}$$

After the corrections are made to the TransformGroups corresponding to the dependent variables, the displacement mismatch is measured again, and the procedure is repeated to convergence.

The following is a code outline for the procedure. `IncrementTG(T, dz)` is a `FourBarLinkage` method that rotates the angle corresponding to `TransformGroup, T` a small angle `dz`. The `getXYVector()` method is described in sub-section 3.2. The procedure within the do loop is repeated until the measured mismatch is sufficiently small. In the following listing, the step numbers refer to steps described in section 3.0.

```
public void solve() {
    do  {
//  step 3a - get mismatch:
    mismatch.set(PinD.getXYVector(1.0, zero));
//  step 3b for angles B and C.
    double s = 100.0;  // scale for the angle
//  compute deltaDxb and deltaDyb and put them
//  into the first column of the Jacobian
//  step 3b for angle B
    IncrementTG(B, 1.0/s);
//  step 3c for angle B
    Jacobian.setColumn(0, PinD.getXYVector(s,
       mismatch));
    IncrementTG(B, -1.0/s);
//  compute deltaDxc and deltaDyc and put them
//  into the second column of the Jacobian
//  step 3b for angle C
    IncrementTG(C, 1.0/s);
//  step 3c for angle C
    Jacobian.setColumn(1, PinD.getXYVector(s,
       mismatch));       // units: m/radian
    IncrementTG(C, -1.0/s);
//  invert the Jacobian in place
    Jacobian.invert();   // units: radian/m
//  then solve for dB and dC
    mismatch.negate();   // units: m
//  calculate correction
    correction/*radians*/.mul(Jacobian
       /*radians/m*/, mismatch/*m*/);
//  correction now is dB, dC in radians
//  step 4
    IncrementTG(B, correction.getElement(0));
    IncrementTG(C, correction.getElement(1));
//  Check the result
}   while(mismatch.normSquared()>1.0e-10);
}
```

3.2 The Constraint Class

Because the `Constraint` class extends Java 3D™ class `Group`, it can be mounted on a scene graph in a particular location. It has, as a member called `MatingPart`, a reference to a node on another part to which it is to be constrained. Along with the usual array of utility methods, it has a `getNetTransform()`, which returns, as a the `Transform3D`, the net transformation of points in the coordinate system of the `MatingPart` node with respect to the coordinate system of the Constraint instance. The method exploits the Java 3D™ Node method `getLocalToVWorld(t)`, which places in parameter t, the net transform from the root to this Node. The core of the method follows:

```
public Transform3D getNetTransform()    {
//  get transform from this to the root
    Transform3D refT3D = new Transform3D();
    this.getLocalToVworld(refT3D);
//  get transform from MatingPart to root
    Transform3D mpT3D = new Transform3D();
    MatingPart.getLocalToVworld(mpT3D);
//  transform from root to this
    refT3D.invert();
//  transform from MatingPart to this
    refT3D.mul(mpT3D);
```

```
        return refT3D;
        }
```

Then the model's `solve()` method can obtain the vector displacement of the origin of the `MatingPart` measured in the coordinate frame of the `Constraint` instance with a simple call:

```
Vector3d v = new Vector3d();
PinD.getNetTransform().get(v);
```

Because the mechanism's `solve()` method uses the Java 3D™ vecmath library classes `GVector` and `GMatrix` for calculation of the updates, it makes sense to offer returns in this form too. In the example planar four bar linkage, the `PinD` has two constraint displacements in the x and y directions, and the following method returns columns of the Jacobian, scaled by `s` as a `Gvector` (the parameter `mismatch` is the displacement `GVector` calculated from the initial guess):

```
public GVector getXYVector(double s, Gvector
    mismatch)   {
    Vector3d v = getDisplacementVector();
    GVector ret = new GVector(2);
    ret.setElement(0, s*(v.x  -
       mismatch.getElement(0)));
    ret.setElement(1, s*(v.y -
       mismatch.getElement(1)));
//  return the Gvector
    return ret;
}
```

3.3 The MachineController

Section 2.3 described the way that the standard Java 3D™ Interpolators work, why they were unsuitable for the present purpose, and the approach to fixing them. This section describes the `MachineController`, which implements that approach. The `MachineController` extends `Behavior`, and it mimics the standard `Interpolator` behavior but with its own clock. First, it has the fields required to create and maintain an independent simulation clock:

```
private long simulationClock;
private int time_step;
private WakeupOnElapsedTime yawn;
```

and second, the `initialize()` and the `processStimulus()` methods required by the `Behavior` class:

```
public void initialize()    {
    simulationClock = 0;
    time_step = 45;   // milliseconds
    yawn = new WakeupOnElapsedTime(time_step);
    wakeupOn(yawn);
    [FourBarLinkage instance].initialize();
    …}
public void processStimulus(Enumeration e)  {
//  do the required housekeeping
    simulationClock += time_step;
    wakeupOn(yawn);
//  then call
    [FourBarLinkage instance].solve()
    …}
```

By exercising control over the advance of the simulation clock at the same time that the simulation model is advanced, this class ensures that successive model states are sufficiently close, ensuring in turn, the stability and convergence of the iterative solution of the constraint conditions.

4.0 SUMMARY

This paper proposed a general approach to 'closing the loop' on a scene graph to support solution of inverse kinematic constraints, but it used specialized methods to solve a particular example problem. While the general approach may be applicable to a broad spectrum of fields, the detailed methods must be adapted to special field. For example, analytical (as opposed to numerical) methods offer high performance in computer animation but may require special kinematic structure [4]. Where the Jacobian is used to express the relationship between dependent variables and differential displacements at constraints, preferences for solution methods depend on the need for speed balanced against the accuracy requirements, the system redundancy, and the potential for the existence of singularities. Welman [5] describes methods based on the cheaper transpose of the Jacobian and a complementary heuristic approach that seeks to minimize displacement mismatches by varying one dependent variable at a time. The Java 3D API gives high-level access to many performance-tuned geometric methods, but also restricts user access to some details of the scene graph on which established solver methods rely.

Mechanism animations using the techniques described in this paper may be viewed as Java applets at VMech.com.

REFERENCES

[1] McCarthy, J. M., Introduction to Theoretical Kinematics, The MIT Press, Cambridge, MA, 1990, ISBN 0-262-13252-4

[2] Press, et al.,Numerical Recipes in C, (http://www.nr.com) The Art of Scientific Computing, Second Edition, Cambridge University Press, 1992.

[3] Sowizral, Henry, Kevin Rushforth, Michael Deering, The Java™ 3D API Specification (http://www.javasoft.com/products/java-media/3D/forDevelopers/J3D_1_2_API/j3dguide/index.html), Addison-Wesley, Reading Massachusetts, 1995 ISBN 0-201-32576-4

[4] Tolani, Deepak, Ambarish Goswami, Norman I. Badler, Real-time inverse kinematics techniques for anthropomorphic limbs,(http://hms.upenn.edu/software/ik/ikan_gm.pdf)Graphical Models, Vol. 62, No. 5, September 2000 Pg 353-388

[5] Welman, Chris, Inverse kinematics and geometric constraints for articulated figure manipulation, Master of Science Thesis, School of Computing Science, Simon Fraser University, September 1993 (available via ftp as a WinZipped PostScript file here (ftp://fas.sfu.ca/pub/cs/theses/1993/ChrisWelmanMSc.ps.gz))

WEBLIOGRAPHY

[a] Yi Zhang with Susan Finger and Johannie Behrens, Introduction to Mechanisms (http://www.cs.cmu.edu/~rapidproto/mechanisms/chpt1.html), Carnegie Mellon University, course notes 39-245 Design through Virtual and Physical Prototyping, Spring 1997.

[b] Bill Baxter, Fast Numerical Methods for Inverse Kinematics, (http://www.cs.unc.edu/~baxter/courses/290/html/img0.htm) Course notes, Comp 290-72, University of North Carolina, Department of Computer Science, February, 2000.

[c] Michael Wagner, Advanced Animation Techniques in VRML 97 (http://vienna.eas.asu.edu/~wagner/academic/vrml98/), Arizona State University,

[d] Sebastian Grassia, Inverse Kinematics (http://www.cs.cmu.edu/afs/cs.cmu.edu/user/spiff/www/physmod95/siggraph95-slides.pdf), SIGGRAPH 95 slides, Physically Based Modeling, Carnegie Mellon University, School of Computer Science.

[e] Introduction to Inverse Kinematics,

[f] Jianmin Zhao and Norman I. Badler, Inverse kinematics positioning using non-linear programming for highly articulated figures, ACM Transactions on Graphics, Volume 13, Issue 4 (1994)

OTHER SOURCES

[a] H-ANIM, Humanoid Animation Working Group, Specification.

[b] IKAN, Inverse Kinematics using ANalytical Methods, Norman I. Badler, University of Pennsylvania.