VR Pinocchio

Studienarbeit

Vorgelegt von
Katharina Kranzdorf

Institut für Computervisualistik
Arbeitsgruppe Computergraphik

Betreuer und Prüfer:
Prof. Dr.-Ing. Stefan Müller

November 2003

(Platzhalter für Aufgabenstellung)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

1 Einleitung

Der alte Gepetto wünscht sich nichts sehnlicher als einen kleinen Jungen. In einer wundervollen, sternklaren Nacht erfüllt eine gute Fee seinen Wunsch, indem sie die kleine Holzmarionette Pinocchio zum Leben erweckt.

In dieser Arbeit macht sich Pinocchio auf zu neuen Abenteuern. Er bekommt wieder Fäden und lebt in einer virtuellen Umegbung.

Im Rahmen dieser Arbeit soll eine Marionette modelliert werden und diese soll sich mit Hilfe inverser Kinematik bewegen. Hinzu kommt eine Steuerung, damit der Benutzer mit ihr interagieren kann wie mit einer realen Marionette.

Meine Motivation, zu diesem Thema eine Studienarbeit zu schreiben war, dass ich während meines bisherigen Studiums Interesse an 3D-Graphik und insbesondere an Modellierung und Animation entwickelt habe. Außerdem führt die Verwendung von Szenegraphen, die ein wesentlicher Bestandteil dieser Arbeit sind, unmittelbar zu visuellen Ergebnissen, was ich als sehr motivierend empfinde.

Meine Entscheidung für Java3D rührt daher, dass mir die Programmiersprache Java besser gefällt und ich darin einfach besser bin als in C bzw. C++. Ich habe auch einige Programmierversuche in C unternommen, aber Java war mir immer sympatischer. Ein weiterer Punkt ist der, dass in der Computergraphik C ein fester Bestandteil ist. Ich sah es als Herrausforderung an, dem ein bißchen entgegen zu wirken und zu zeigen, dass man auch mit Java zu dem gleichen Ergebnis kommt.

Zum Aufbau der Arbeit: Im folgenden Abschnitt stelle ich drei Methoden zur Umsetzung inverser Kinematik vor. Daran schließt sich eine kurze Einführung in Java3D an. In Kapitel 4 erläutere ich die Modellierung der Marionette. Im darauffolgenden Kapitel wird die Umsetzung der inversen Kinematik erklärt. Abschließend beschreibe ich noch die Steuerung der Marionette.

2 Inverse Kinematik

Die inverse Kinematik ist ein rein mathematisches Modell zur Simulation von Bewegungsabläufen. Der Ansatz stammt aus der Robotik, die schon früh mit dem Problem der richtigen Positionierung von Greifarmen und Werkzeugen konfrontiert war. In der Computeranimation dient die inverse Kinematik zur Simulation der mechanischen Bewegungsabläufe des menschlichen Skeletts.

Die Vorgehensweise läßt sich mit wenigen Worten beschreiben: Bei der inversen Kinematik werden die gewünschte Position und die mögliche Orientierung des Endeffektors vorgegeben. Die benötigten Winkel der Gelenke werden berechnet.

An dieser Stelle möchte ich eine kurze Definition des Begriffes *Endeffektor* liefern. In der gesamten Literartur wird dieser Begriff im Zusammenhang mit Inverser Kinematik gebraucht. Leider konnte ich keine genaue Definition dazu finden.

Definition: Der Endeffektor ist das letzte Glied einer Gelenkkette.

Das Problem kann keine, eine oder viele Lösungen haben. Wenn die Konfiguration so viele Beschränkungen hat, dass es keine Lösung gibt, so nennt man dieses System *überbeschränkt*. Falls es nur wenige Einschränkungen auf dem System gibt und viele Lösungen existieren, dann nennt man das System *unterbeschränkt*. Der *erreichbare Arbeitsraum* (reachable workspace) ist das Volumen, das der Endeffektor mit mindestens einer Orientierung erreichen kann. Der *vollmanipulierbare Arbeitsraum* (dextrous workspace) ist das Volumen, das der Endeffektor in jeder beliebigen Orientierung erreichen kann.

Im Folgenden werde ich Methoden der inversen Kinematik vorstellen. Die sich anschließenden Abschnitte 2.1 und 2.2 sind aus dem Buch „Computer Animation, Algorithms and Techniques“ [3] von Rick Parent entnommen.

2.1 Einfache IK Probleme analytisch lösen

Bei besonders einfachen Mechanismen können die Gelenkwinkel der gewünschten Position analytisch bestimmt werden. Dazu betrachtet man die Geometrie der Verbindung. Man stelle sich einen einfachen zweigliedrigen Arm in einem zweidimensionalen Raum vor. Die Länge des ersten Glieds sei L_1und L_2 die Länge des zweiten Glieds. Wenn das erste Glied am Ausgangspunkt fest verankert ist, dann kann jede Position jenseits von $|L_1 - L_2|$ erreicht werden und jede Position innerhalb von $L_1 + L_2$ (s. Abb. 1).

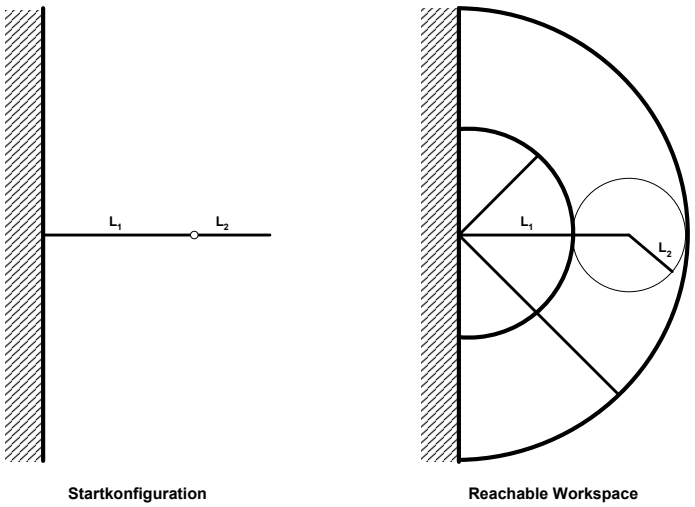


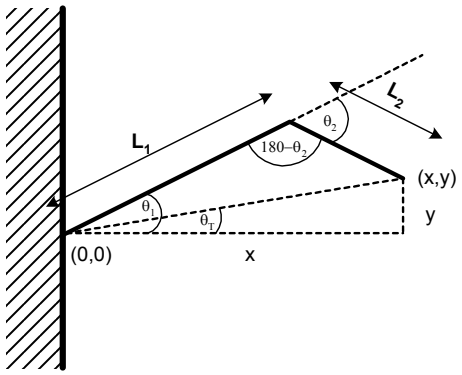
Abbildung 1: Eine einfache Verbindung

Angenommen (ohne Beschränkung der Allgemeinheit) der Ausgangspunkt liegt im Nullpunkt. Bei einem einfachen Problem gibt der Benutzer die (x,y) Koordinaten der gewünschten Position des Endeffektors an. Die Gelenkwinkel, θ_1 und θ_2, können berechnet werden, indem man den Abstand vom Ausgangspunkt zum Zielpunkt bestimmt und mit Hilfe der Cosinusregel die inneren Winkel berechnet (siehe Abbildung 2, Gleichungen 3 und 5). Wenn erst einmal die inneren Winkel berechnet sind, können die Rotationswinkel der beiden Glieder bstimmt werden. Der erste Schritt ist natürlich sicherzustellen, dass die Zielposition innerhalb des erreichbaren Raums des Endeffektors liegt. Dies stellt man folgendermaßen sicher: $L_1 - L_2 \leq \sqrt{x^2 + y^2} \leq L_1 + L_2$.

In diesem einfachen Beispiel gibt es nur 2 Lösungen; die Lösungskonfiguration ist symetrisch zur Geraden von $(0, 0)$ nach (x, y). Dies zeigt sich in den Gleichungen in Abbildung 2, da der Arcus-Cosinus zu einem gegebenen Wert 2 Winkel erzeugt, einen positiven und ein negativen. Für kompliziertere Konfigurationen gibt es unendlich viele Lösungen, die die gewünschte Position des Endeffektors bestimmen.

Die meisten Mechanismen in der Computeranimation sind zu komplex um analytisch gelöst zu werden. Bei komplizierten Mechanismen kann man die Bewegung inkrementell konstruieren. Zu jedem Zeitpunkt wird eine Berechnung ausgeführt, die die bestmögliche Änderung jedes Gelenkwinkels

ermittelt, um den Endeffektor in seine gewünschte finale Position und Orientierung zu bringen. Im folgenden werden 2 inkrementelle Methoden zur Lösung eines inverse Kinematik Problems vorgestellt.



$$\cos(\theta_T) = \frac{x}{\sqrt{x^2+y^2}} \tag{1}$$

$$\theta_T = \arccos\Big(\frac{x}{\sqrt{x^2+y^2}}\Big) \tag{2}$$

$$\cos(\theta_1-\theta_T) = \frac{L_1^2+x^2+y^2-L_2^2}{2L_1\sqrt{x^2+y^2}} \tag{3}$$

$$\theta_1 = \arccos\Big(\frac{L_1^2+x^2+y^2-L_2^2}{2L_1\sqrt{x^2+y^2}}\Big)+\theta_T \tag{4}$$

$$\cos(180-\theta_2) = \frac{L_1^2+L_2^2-(x^2+y^2)}{2L_1L_2} \tag{5}$$

$$\theta_2 = \arccos\Big(\frac{L_1^2+L_2^2-(x^2+y^2)}{2L_1L_2}\Big) \tag{6}$$

Abbildung 2: Lösung eines einfachen inverse Kinematik Problems

2.2 Die Jacobi Matrix

Die Methode ist eine sehr mathematische. Sie basiert auf Funktionen und deren partiellen Ableitungen. Auf den mathematischen Hintergrund möchte ich an dieser Stelle nicht weiter eingehen, da dieser zur Lösung des Problems nicht relevant ist.

Die Gleichung, nach der das IK-Problem gelöst werden soll, lautet wie

folgt:

$$V = J(\theta) \cdot \dot{\theta} \tag{7}$$

V ist ein Vektor der linearen und der Rotationsgeschwindigkeiten und repräsentiert die gewünschten Änderungen im Endeffektor. Diese basiert auf der Differenz zwischen der aktuellen Position/Orientierung des Endeffektors zur gewünschten Zielposition/-orientierung. Die Geschwindigkeiten sind Vektoren im 3D-Raum. Jede Geschwindigkeit hat eine x, y und z Komponente (siehe Gleichung 8). v bezeichnet die Position und ω die Orientierung.

$$V = [v_x, v_y, v_z, \omega_x, \omega_y, \omega_z] \tag{8}$$

$\dot{\theta}$ sind die Gelenkwinkelgeschwindigkeiten, die die Unbekannten in der Gleichung darstellen.

$$\dot{\theta} = [\dot{\theta}_1, \dot{\theta}_2, \dot{\theta}_3, ..., \dot{\theta}_n]^T \tag{9}$$

J ist die Jacobimatrix. Diese ist eine Funktion der aktuellen Lage der Konfiguration (siehe Gleichung 10). Jeder Term der Jacobimatrix bringt die Veränderung eines konkreten Gelenks in Beziehung zu einer spezifischen Veränderung des Endeffektors.

$$\begin{pmatrix} \frac{\delta v_x}{\delta \theta_1} & \frac{\delta v_x}{\delta \theta_2} & \cdots & \frac{\delta v_x}{\delta \theta_n} \\ \frac{\delta v_y}{\delta \theta_1} & \frac{\delta v_y}{\delta \theta_2} & \cdots & \frac{\delta v_y}{\delta \theta_n} \\ \cdots & \cdots & \cdots & \cdots \\ \frac{\delta \omega_z}{\delta \theta_1} & \frac{\delta \omega_z}{\delta \theta_2} & \cdots & \frac{\delta \omega_z}{\delta \theta_n} \end{pmatrix} \tag{10}$$

Die Rotationsveränderung des Endeffektors ω ist die Winkelgeschwindigkeit um die Rotationsachse am betrachteten Gelenk. Die lineare Veränderung des Endeffektors ist das Kreuzprodukt der Drehachse (*axis of revolution*) und dem Vektor des aktuellen Gelenks zum Endeffektor.

Die gesuchte Winkel- und Lineargeschwindigkeit wird berechnet durch den Unterschied der aktuellen Konfiguration zur gewünschten Konfiguration des Endeffektors.

Das Problem ist, die beste Linearkombination, die sich aus den vielen verschiedenen Gelenken ergibt, zu finden um daraus die gewünschten Geschwindigkeiten des Endeffektors zu bestimmen.

Wichtig ist, dass alle verwendeten Koordinaten im selben Koordinatensystem liegen. Oftmals sind gelenkspezifische Koordinaten in lokalen Koordinatensystemen, daher muss sicher gestellt werden, dass diese Koordinaten in ein globales Koordinatensystem umgewandelt werden.

2.2.1 Ein einfaches Beispiel in 2D

Abbildung 3 zeigt ein einfaches Beispiel mit 3 Gelenken. Der Endeffektor E soll zur Zielposition G bewegt werden. Die Drehachse liegt senkrecht zu den Gelenken also bei $(0, 0, 1)$. Der Eindruck einer inkrementellen Rotation g_i kann durch ein Kreuzprodukt der Drehachse und dem Vektor vom aktuellen Gelenk zum Endeffektor V_i bestimmt werden (siehe Abbildung 4). Die gewünschte Veränderung des Endeffektors ist die Differenz der aktuellen Position des Endeffektors und der Zielposition (siehe Gleichung 11).

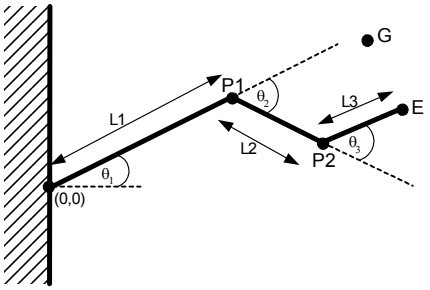
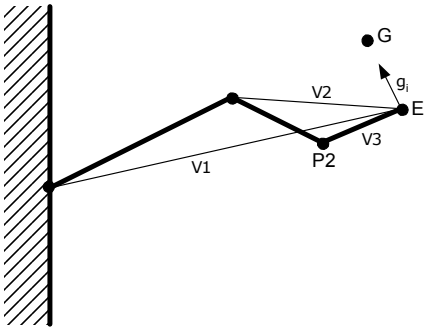


Abbildung 3: Eine ebene Verbindung mit 3 Gelenken



$$
\begin{aligned}
V_1 &= E - P_0 \\
V_2 &= E - P_1 \\
V_3 &= E - P_2
\end{aligned}
$$

Abbildung 4: Momentane Änderung der Position veranlasst durch die Gelenkrotation

$$\begin{pmatrix} (G_x - E_x) \\ (G_y - E_y) \\ (G_z - E_z) \end{pmatrix}$$

$$= \begin{pmatrix} ((0,0,1) \times E_x) & (0,0,1) \times (E_x - P_1) & (0,0,1) \times (E_x - P_2) \\ ((0,0,1) \times E_y) & (0,0,1) \times (E_y - P_1) & (0,0,1) \times (E_y - P_2) \\ ((0,0,1) \times E_z) & (0,0,1) \times (E_z - P_1) & (0,0,1) \times (E_z - P_2) \end{pmatrix} \quad (11)$$

$$\begin{pmatrix} \dot{\theta}_1 \\ \dot{\theta}_2 \\ \dot{\theta}_3 \end{pmatrix}$$

Um die Gelenkgeschwindigkeiten bestimmen zu können, muß die Gleichung nach seinen Unbekannten $\dot{\theta}$ aufgelöst werden:

$$\dot{\theta} = J^{-1}V \quad (12)$$

Das Problem, das hierbei entsteht ist die inverse Jacobimatrix. Diese läßt sich einfach errechnen, wenn es sich um eine quadratische Matrix handelt. Wenn nun die inverse Matrix J^{-1} nicht existiert, handelt es sich um ein singuläres System. Eine Singularität tritt auf, wenn es keine Linearkombination gibt, um die gewünschten Endeffektorgeschwindigkeiten zu erreichen.

In dem Fall, dass die Jacobimatrix keine reguläre quadratische Matrix ist, existiert keine konventionelle Inverse. In diesem Fall benutzt man die *Pseudoinverse* J^+. Diese berechnet sich wie folgt:

$$\begin{aligned} V &= J\dot{\theta} && (13) \\ J^T V &= J^T J\dot{\theta} && (14) \\ (J^T J)^{-1} J^T V &= (J^T J)^{-1} J^T J\dot{\theta} && (15) \\ J^+ V &= \dot{\theta} && (16) \end{aligned}$$

$(J^T J)^{-1} J^T$ bezeichnet man als die Pseudoinverse J^+ und $(J^T J)^{-1} J^T J$ ist die Einheitsmatrix. Da ein Vektor multipliziert mit der Einheitsmatrix keine Veränderung erfährt, kann man diese wegkürzen.

2.2.2 Probleme bei der Jacobimethode

Bei der Berechnung des obengenannten Beispiels stößt man auf ein Problem: Die Berechnung der Inversen. In Gleichung 11 sieht man, dass es sich um eine quadratische Matrix handelt. Die Bestimmung der Inversen scheint einfach zu sein. Doch bei der Berechnung der Determinanten kommt stets 0 heraus, d.h. es handelt sich nicht um eine reguläre Matrix. Somit ist die Berechnung

der Inversen nicht möglich! Die Lösung für dieses Problem ist ganz einfach: Die Z-Komponenten in der Jacobimatrix sind immer 0. Deshalb kann man diese weglassen, und dadurch ist es möglich die Pseudoinverse zu bestimmen. Mit diesem kleinen Trick läßt sich also eine Inverse berechnen.

Ein weiteres Problem dieser Methode liegt in den Geschwindigkeiten. Um diese anwenden zu können muß man Zeiteinheiten wählen. Auf diese Zeiteinheiten wird nun die Geschwindikeit angewendet. Dabei kann es passieren, dass der Endeffektor den Zielpunkt überschreitet. Dieses Verhalten kann endlos so weiter gehen. Somit ist es möglich, dass der Zielpunkt vom Eneffektor zwar erreicht werden kann, dieser aber in springenden Bewegungen um den Zielpunkt kreist, ihn also niemals erreicht. Auch wenn der Zielpunkt ausserhalb der Reichweite des Endeffektors liegt gibt es Probleme. Man würde erwarten, dass der Algorithmus den Endeffektor an dem Punkt platziert, der in Position und Richtung dem Zielpunkt am ähnlichsten ist. Dies ist aber nicht der Fall. Der Algorithmus positioniert den Endeffektor scheinbar wahllos in seinem erreichbaren Arbeitsraum.

Als Quelle für den nachfolgenden Abschnitt habe ich den Artikel „Making Kine More Flexible“ [2] von Jeff Lander herangezogen.

2.3 Die Cyclic-Coordinate-Descent (CCD) Methode

Diese Technik wurde zuerst behandelt von Li-Chun Tommy Wang und Chih Cheng Chen in einem Artikel in „IEEE Transaction on Robotics and Automation“.

Die Methode startet beim letzten Glied in der Gelenkkette und arbeitet rückwärts jedes Glied in der Kette ab (siehe Abbildungen 5 bis 8). Man

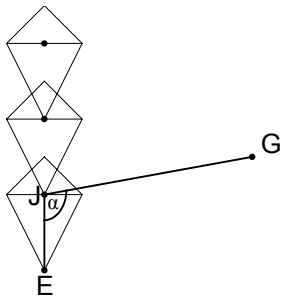


Abbildung 5: 1. Schritt: Berechnen der Vektoren

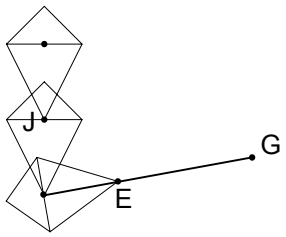


Abbildung 6: 2. Schritt: Drehen des Endeffektors

beginnt mit dem letzten Gelenk in der Kette. Man berechnet den Vektor

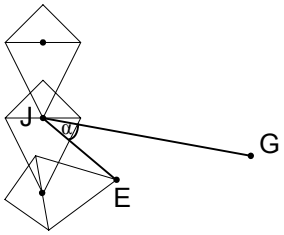


Abbildung 7: 1. Schritt: Berechnen der Vektoren

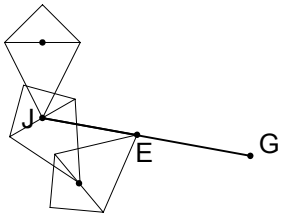


Abbildung 8: 2. Schritt: Drehen des Endeffektors

vom aktuellen Gelenk J zum Endeffektor E und den Vektor vom aktuellen Gelenk J zum Zielpunkt G. Zwischen diesen beiden Vektoren berechnet man den Winkel α, indem man das Skalarprodukt bildet (siehe Abbildung 5). Anhand des Kreuzproduktes der beiden Vektoren, kann entschieden werden in welche Richtung gedreht werden muß. Das Kreuzprodukt liefert einen Vektor der senkrecht auf den beiden steht. Im 2D kann man dann anhand der z-Komponenete des Kreuzproduktes entscheiden, ob das Gelenk im oder gegen der Uhrzeigersinn gedreht wird. Im 3D ist das leider nicht ganz so trivial. Darauf gehe ich später noch genauer ein. Um diesen Winkel α wird das Glied dann gedreht. Das Ergebnis kann man in Abbildung 6 sehen. Jetzt geht man zum nächsten Gelenk in der Kette und wiederholt den Prozess wie man in den Abbildungen 7 und 8 sehen kann.

So geht man alle Gelenke in der Kette durch. Hat man das erste Glied erreicht beginnt man wieder beim letzten. Abgebrochen wird der Prozess, wenn der Zielpunkt nahezu erreicht ist oder eine vorgebene Anzahl von Iterationen erreicht ist.

2.3.1 Vorteile der CCD-Methode

Die Mathematik der CCD-Methode ist sehr einfach. Auch der daraus entstandene Algorithmus ist einfach zu implementieren. Das resultierende Ergebnis ist sehr zufriedenstellend. Alle Probleme, die bei der Jacobimethode auftreten können, gibt es hier nicht.

Nach diesem Vergleich der beiden Algorithmen habe ich mich aus oben genannten Gründen für die CCD-Methode entschieden.

3 Java3D

Die Java3D API ist eine Schnittstelle für die Darstellung und Interaktion von 3D Graphik. Java3D ist eine Standarderweiterung zum Java 2 SDK. Die API bietet eine Sammlung von High-Level-Konstrukten zur Erstellung und Manipulation von 3D Geometrien und Strukturen um diese zu rendern.

3.1 Einen Szenegraph erstellen

Die zentrale Struktur in Java3D ist der Szenegraph. In diesem werden visuelle und auditive Objekten und deren Eigenschaften gespeichert. Ganz allgemein ist ein Graph eine Datenstruktur von Knoten und Übergängen. Ein Knoten ist ein Datenelement und ein Übergang ist eine Beziehung zwischen Datenelementen. Die Knoten im Szenegraph sind Instanzen von Java3D-Klassen (siehe Abbildung 9).

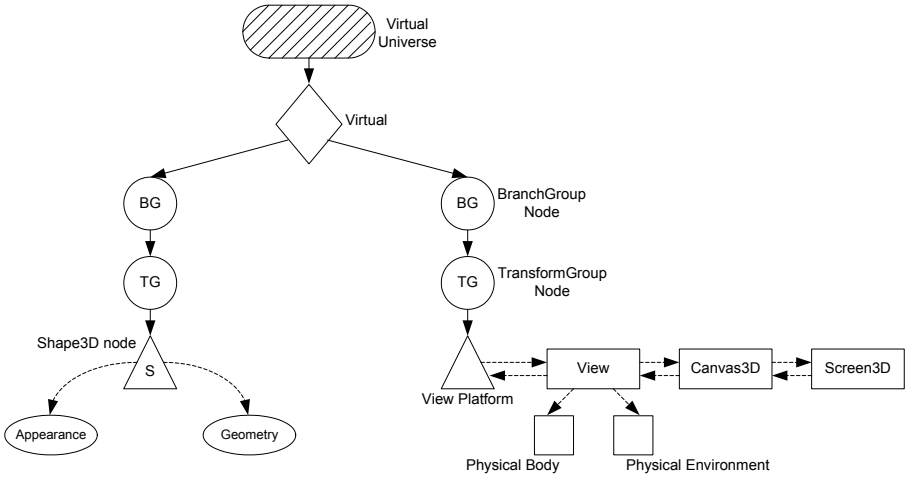


Abbildung 9: Szenegraph

VirtualUniverse: Dies ist der Ursprung einer jeden Szene. Es enthält alle notwendigen Objekte für die 3D-Darstellung der Szene. Alle Objekte befinden sich in einem Rechtssystem.

Locale: Locales dienen als Container für Subgraphen, welche dann wieder über einen BranchGroup-Knoten dem Locale hinzugefügt werden. Sie spalten den Szenegraphen in einen *Objektzweig* und einen *Blickzweig*.

BranchGroup: Branchgroups sind Wurzeln eigenständiger Teilgraphen.

TransformGroup: Eine TransformGroup beschreibt eine affine Abbildung und wendet diese auf alle seine Nachfolge Objekte an. Die Lokalen Koordinatensysteme ergeben sich durch die Multiplikation aller Transformationen auf dem Weg zum Virtual Universe.

Shape3D: Dies sind die Blätter des Baumes. Es handelt sich hierbei um einfache geometrische Objekte wie Kugeln, Quader, etc. Jedem Objekt wiederum können Attribute (*NodeComponents*) über die Art des Aussehens und die Art der zugrunde liegenden Geometrie zugeordnet werden.

ViewPlatform: Die ViewPlatform ist eine virtuelle Platform auf der der Benutzer steht.

View: Das View Objekt fasst die Informationen zusammen, die zur Projektion des Bildes notwendig sind. Es dient als Vermittler zwischen realer und virtueller Welt.

PhysicalBody: Der PhysicalBody beschreibt die notwendigen pysikalischen Daten eines Benutzers wie z.B. Blickrichtung und die Position der Augen.

PhysicalEnvironment: Das PhysicalEnvironment enthält Kalibrierungsinformationen der realen Welt, wie z.B. Information über Eingabegeräte (Datenhandschuh und ähnliches) oder Position von Lautsprechern.

Canvas3D: Das Canvas3D ist die Zeichenfläche, auf der der Java3D-Renderer die Objekte ausgibt.

Screen3D: Der Screen3D enthält Informationen über die Darstellungsgeräte, wie z.B. HMD, Cave, etc.

Man unterscheidet zwei verschiedene Arten von Beziehungen: Die *Eltern-Kind-Beziehung*, dargestellt durch einen durchgezogenen Pfeil, und eine *Reference*, dargestellt durch einen gestrichelten Pfeil, siehe Abbildung 9. Ein Gruppenknoten (TransformGroup oder BranchGroup) kann viele Kinder haben, aber jedes Kind kann nur einen Elternknoten besitzen. Ein Blatt kann keine Kinder haben. Eine Referenz assoziiert ein NodeComponent-Objekt mit einem Szenegraph-Knoten. NodeComponents und References sind kein Teil des Szenegraphen.

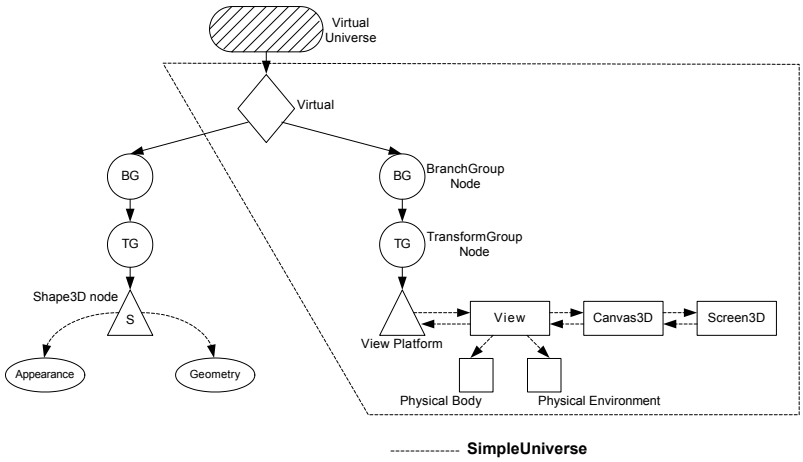


Abbildung 10: Klasse: SimpleUniverse

Java stellt eine Klasse `SimpleUniverse` zur Verfügung. Diese erzeugt den gesamten Blickzweig mit sinnvollen Voreinstellungen, siehe Abbildung 10. Dies ist für einfache Anwendungen eine nützliche Erleichterung.

Das folgende Codefragment zeigt den Konstruktor meiner Klasse `Marionette`. Es zeigt die einfache Erstellung eines SimpleUniverse. Die Methode `createSceneGraph` (Zeile 5) erstellt den Szenegraph der Marionette und wird in Kapitel 4 näher beschrieben.

```
1  public Marionette() {
2    gc = SimpleUniverse.getPreferredConfiguration();
3    canvas3D = new Canvas3D(gc);
4    canvas3D.setSize(512, 512);
5    scene = createSceneGraph();
6    scene.compile();
7    universe = new SimpleUniverse(canvas3D);
8    universe
9      .getViewingPlatform()
10     .setNominalViewingTransform();
11   universe.addBranchGraph(scene);
12 }
```

Im weiteren befasse ich mich noch oberflächlich mit Licht, Material und Texturen.

Java3D benutzt ein Lichtmodell, das dem physikalischen Licht der realen Welt sehr nahen kommt. Es unterstützt ambientes, diffuses und spekulares Licht. Man kann unter vier Lichtquellen auswählen: eine Umgebungslichtquelle(*ambient light*), eine gerichtete (*directional light*) Lichtquelle, eine Punktlichtquelle (*point light*) und eine Spotlichtquelle (*spot light*). Ich habe mich in meiner Anwendung für das Punktlicht entschiden.

```
PointLight(
  Color3f color,
  Point3f position,
  Point3f attenuation)
```

Im Konstruktor `PointLight` gibt man die Farbe des Lichtes an, die Position, die es in der Szene haben soll und die Abschwächung des Lichtes. Diese kann konstant, linear oder quadratisch sein.

Damit das Licht auch irgendeine Auswirkung hat, muß jedes zu beleuchtende Objekt ein Material haben.

```
Material(
  Color3f ambientColor,
  Color3f emissiveColor,
  Color3f diffuseColor,
  Color3f specularColor,
  float shininess)
```

Im Konstruktor der Klasse `Material` gibt man die Eigenschaften des Materials an.

Bei den Texturen habe ich nur mit der Klasse `Texture2D` gearbeitet.

```
Texture2D(
  int mipMapMode,
  int format,
  int width,
  int height,
  int boundaryWidth)
```

Im Konstruktor gibt man auch hier die Eigenschaften der Textur an. Ein Problem, dass bei den Texturen aufgetreten ist, war die Reflexion des Lichtes. Die Textur selbst kann kein Licht reflektieren. In der Standardeinstellung ersetzt die Testur das Material des Objektes vollständig. Deshalb

muß man noch ein Texturattribut `MODULATE` setzen. Dieses moduliert die Materialeigenschaften mit der Textur.

Mehr zu Java3D findet man unter:
http://java.sun.com/products/java-media/3D/collateral/ ([1]).

4 Modellierung der Marionette

In diesem Abschnitt werde ich den Szenegraphen der Marionette vorstellen. An ausgewählten Codebeispielen werde ich zeigen wie man den Szenegraphen in Java implementiert.

Abbildung 11 zeigt den Szenegraph der Marionette. Direkt unter dem SimpleUniverse hängt die BranchGroup `bgRoot`. Diese wird am Ende der Methode `createSceneGraph` zurückgegeben. Im untenstehenden Codefragment wird gezeigt wie man TransformGroups erzeugt und diese dann in den Szenegraphen hängt.

Abbildung 11: Szenegraph der gesamten Marionette

```
    // Marionette rechts links bewegen
    tgTranslateMarionetteX = new TransformGroup(translate);
    tgTranslateMarionetteX.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);

    // Marionette hoch runter bewegen
    tgTranslateMarionetteY = new TransformGroup(translate);
    tgTranslateMarionetteY.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);

    // Rotation der Marionette um y
    rotate.setIdentity();
    tgRotateMarionetteY = new TransformGroup(rotate);
    tgRotateMarionetteY.setCapability(
      TransformGroup.ALLOW_TRANSFORM_WRITE);
    tgRotateMarionetteY.setUserData(JOINT_MARIONETTE_Y);

    // an den Objektknoten wird die Marionette geh{\"a}ngt
    bgRoot.addChild(tgTranslateMarionetteY);
    tgTranslateMarionetteY.addChild(tgTranslateMarionetteX);
    tgTranslateMarionetteX.addChild(bgMarionette);
    bgMarionette.addChild(tgRotateMarionetteY);
```

4.1 Modellierung des Kopfes

Abbildung 12 zeigt den Teilszenegraphen des Kopfes. Dieser Teil des Szenegraphen hängt direkt unter der TransformGroup `tgRotateMarionetteY`. Dieser spielt allerdings später bei der Bewegung der Marionette keine Rolle.

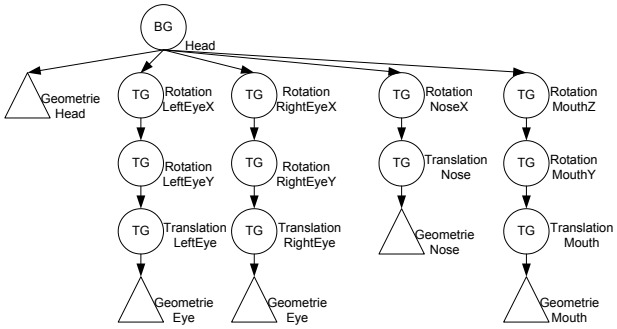


Abbildung 12: Teilszenegraph des Kopfes

Das folgende Codebeispiel zeigt wie die Nase erstellt, an die richtige Stelle transformiert wird und dann in der Szenegraph gehängt wird.

```
1  // Nase
2      Cone nose = new Cone(0.015f, 0.04f, appOrange);
3
4      rotate.rotX(Math.PI / 2);
5      TransformGroup tgRotateNose =
6        new TransformGroup(rotate);
7
8      translate.set(
9        new Vector3d(
10         0,
11         head.getRadius() + 0.5 * nose.getHeight(),
12         0));
13     TransformGroup tgTranslateNose =
14       new TransformGroup(translate);
15
16     bgHead.addChild(tgRotateNose);
17     tgRotateNose.addChild(tgTranslateNose);
18     tgTranslateNose.addChild(nose);
```

4.2 Modellierung der Arme

Die Abbildungen 13 und 14 zeigen die Teilszenegraphen der Arme der Marionette. Diese hängen an der BranchGroup Body.

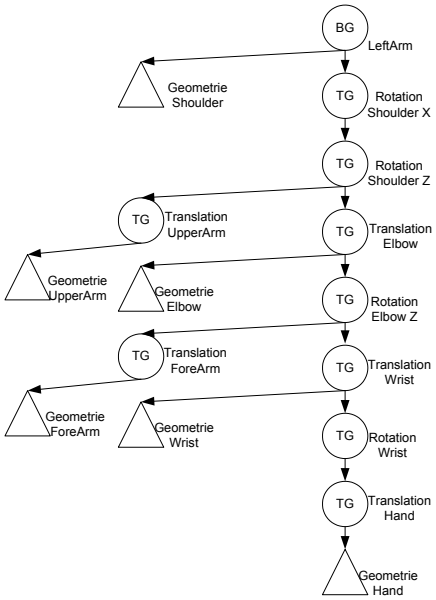


Abbildung 13: Teilszenegraph des linken Arms

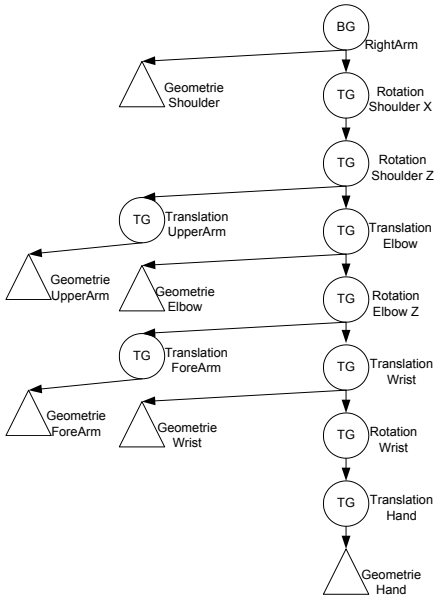


Abbildung 14: Teilszenegraph des rechten Arms

Der folgende Code zeigt den Teil des Szenegraphen, wo die linke Schulter und der linke Oberarm mit all seinen TransformGroups in den Szengraphen gehängt wird. Die Geometrie der Schulter ist eine Kopie der Geometrie `joint`. Alle anderen Gelenke sind auch Kopien von `joint`. Unter der BranchGroup `bgLeftArm` hängen 2 TransformGroups. Eine um den Arm um x drehen zu können und die andere um den Arm auch um z drehen zu können. Der Oberarm wird so transliert, dass er direkt unter der Schulter hängt.

```
1     // Schulter
2     bgLeftArm.addChild(joint.cloneTree(true));
3
4     // Oberarm
5     rotate.setIdentity();
6     tgRotateShoulderLeftX = new TransformGroup(rotate);
7     tgRotateShoulderLeftX.setCapability(
```

```
8         TransformGroup.ALLOW_TRANSFORM_WRITE);
9       tgRotateShoulderLeftX.setUserData(
10        JOINT_SHOULDER_LEFT_X);
11
12      rotate.setIdentity();
13      tgRotateShoulderLeftZ = new TransformGroup(rotate);
14      tgRotateShoulderLeftZ.setCapability(
15        TransformGroup.ALLOW_TRANSFORM_WRITE);
16      tgRotateShoulderLeftZ.setUserData(
17        JOINT_SHOULDER_LEFT_Z);
18
19      translate.set(
20        new Vector3d(
21          upperArm.getRadius() - joint.getRadius(),
22          - (0.5 * upperArm.getHeight() + joint.getRadius()),
23          0));
24      TransformGroup tgTranslateUpperArmLeft =
25        new TransformGroup(translate);
26
27      bgLeftArm.addChild(tgRotateShoulderLeftX);
28      tgRotateShoulderLeftX.addChild(tgRotateShoulderLeftZ);
29      tgRotateShoulderLeftZ.addChild(tgTranslateUpperArmLeft);
30      tgTranslateUpperArmLeft.addChild(
31        upperArm.cloneTree(true));
```

4.3 Modellierung der Beine

Die Abbildungen 15 und 16 zeigen die Teilszenegraphen der Beine der Marionette. Diese hängen auch an der BranchGroup Body.

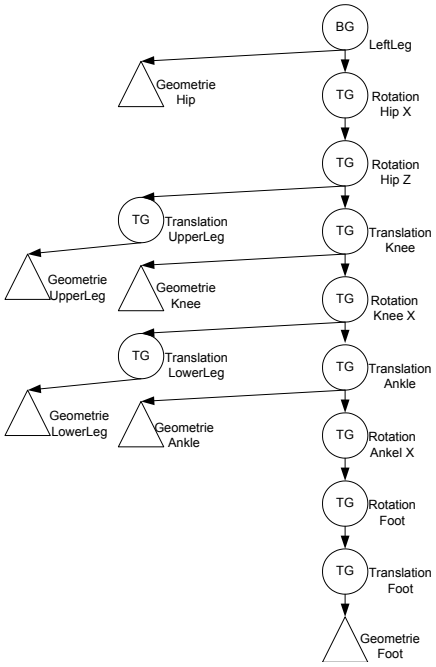


Abbildung 15: Teilszenegraph des linken Beins

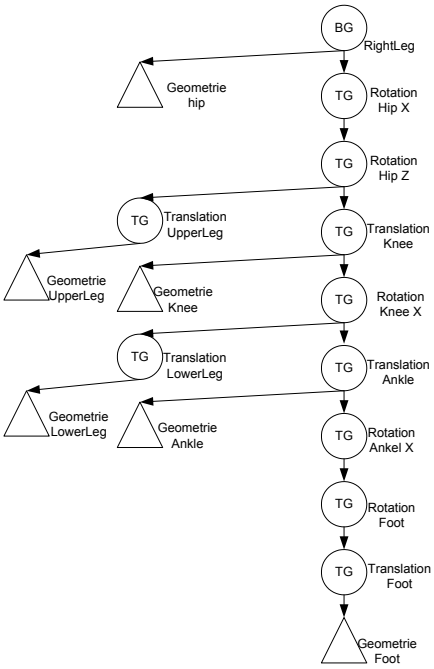


Abbildung 16: Teilszenegraph des rechten Beins

Bei diesem Code ist interessant, dass der Fuß, bevor er an die richtige Stelle transliert wird, ersteinmal in die richtige Position rotiert wird.

```
1     // Fussgelenk
2     translate.set(
3       new Vector3d(
4         0,
5         - (lowerLeg.getHeight() + 2 * joint.getRadius()),
6         0));
7     TransformGroup tgTranslateAnkleLeft =
8       new TransformGroup(translate);
9
10    tgRotateKneeLeftX.addChild(tgTranslateAnkleLeft);
11    tgTranslateAnkleLeft.addChild(joint.cloneTree(true));
```

```
12
13      // Fuss
14
15      // Fuss nach vorne kippen
16      rotate.rotX(Math.PI / 2);
17      TransformGroup tgRotateFootLeft =
18        new TransformGroup(rotate);
19
20      rotate.setIdentity();
21      tgRotateAnkelLeftX = new TransformGroup(rotate);
22      tgRotateAnkelLeftX.setCapability(
23        TransformGroup.ALLOW_TRANSFORM_WRITE);
24      tgRotateAnkelLeftX.setUserData(JOINT_ANKLE_LEFT_X);
25
26      translate.set(
27        new Vector3d(
28          0,
29          0.5 * foot.getHeight() - joint.getRadius(),
30          foot.getRadius() + joint.getRadius()));
31      TransformGroup tgTranslateFootLeft =
32        new TransformGroup(translate);
33
34      tgTranslateAnkleLeft.addChild(tgRotateAnkelLeftX);
35      tgRotateAnkelLeftX.addChild(tgRotateFootLeft);
36      tgRotateFootLeft.addChild(tgTranslateFootLeft);
37      footLeft = (Cylinder) foot.cloneTree(true);
38      footLeft.setUserData(ENDEFFECTOR_FOOT_LEFT);
39      tgTranslateFootLeft.addChild(footLeft);
```

5 Umsetzung der CCD-Methode

In Kapitel 2.3 wurde schon der mathematischen Hintergrund der CCD-Methode beschrieben. Das Beispiel, das in dem oben genannten Artikel von Jeff Lander ([2]) beschrieben wird, ist nur in 2D. In diesem Kapitel möchte ich nun vorstellen wie ich diese Methode in 3D umgestezt und in Java implementiert habe.

Bei der CCD-Methode wird wie bereits erwähnt am Ende jedes Schleifendurchlaufs die Gelenkstellung verändert. Ich wollte allerdings vermeiden, dass man diese Veränderungen, auf dem Weg zum Ziel, sieht. Die Bewegungen der Marionette sollen flüssig sein und nicht abgehackt aussehen. Ich habe das so gelöst, dass ich eine Kopie der Marionette erstelle und auf dieser Kopie die ganzen Berechnungen ausführe. Auf diese Weise werden die Zwischenschritte nicht sichtbar und nur die finalen Gelenkwinkel werden in die Originalpuppe kopiert. Am Ende der Methode `createSceneGraph` in der Klasse `Marionette` wird die Kopie mit dem folgenden Code erstellt:

```
clone = (BranchGroup)(bgMarionette.cloneTree(true));
```

Ab der BranchGroup `bgMarionette` wird der Szenegraph kopiert.

5.1 Die Klasse Joint

In der Klasse `Joint` werden die Gelenke verwaltet. Im Kontruktor der Klasse werden folgende Eigenschaften übergeben:

- der Name des Gelenks,
- der Name der TransformGroups in der Original-Marionette für die Drehungen um x, y und z,
- die Kopie der Marionette und
- ein minimaler und ein maximaler Winkel für jede Drehachse (x, y, z).

Da bei der Berechnung der inversen Kinematik nur auf der Kopie der Marionette gearbeitet wird, muß im Konstruktor die richtige Zuordnung passieren. Jede TransformGroup in der Marionette hat einen Namen. Über diesen Namen kann man in der Kopie die richtigen TransformGroups identifizieren.

Die Methode `getWorldCoordinates()` in der Klasse `Joint` berechnet die Weltkoordinaten von Punkten. Dazu traversiert sie den Szenegraphen und multipliziert die Matritzen. Die Methode `isAxisSupported(int axis)`

überprüft, ob die angegbene Achse in dem Gelenk als Drehachse verwendet werden darf. Die Methoden `getMin(int axis)`, `getMax(int axis)` sowie `getTransformGroup(int axis, boolean original)` holen aus dem Gelenk das Rotationsminimum bzw. -maximum und die gesuchte TransformGroup einer gewünschten Achse.

5.2 Die Klasse Configuration

Um die inverse Kinematik richtig anwenden zu können, braucht man noch die Klasse `Configuration`. In dieser wird die die Reihenfolge der Gelenke festgelegt, wie diese dann die inverse Kinematik durchlaufen. Der folgende Code ist ein Beispiel für die Konfiguration des linken Beins der Marionette.

```
1     configurationLowerLegLeft =
2       new Configuration(lowerLegLeft, clone);
3     configurationLowerLegLeft.addJoint(
4       new Joint(
5         "Marionette",
6         null,
7         tgRotateMarionetteY,
8         null,
9         clone,
10        0,
11        0,
12        ROTATE_MODEL_MIN,
13        ROTATE_MODEL_MAX,
14        0,
15        0));
16    configurationLowerLegLeft.addJoint(
17      new Joint(
18        "HipLeft",
19        tgRotateHipLeftX,
20        null,
21        tgRotateHipLeftZ,
22        clone,
23        ROTATE_HIP_LEFT_X_MIN,
24        ROTATE_HIP_LEFT_X_MAX,
25        0,
26        0,
27        ROTATE_HIP_LEFT_Z_MIN,
28        ROTATE_HIP_LEFT_Z_MAX));
29    configurationLowerLegLeft.addJoint(
30      new Joint(
```

```
31          "KneeLeft",
32          tgRotateKneeLeftX,
33          null,
34          null,
35          clone,
36          ROTATE_KNEE_LEFT_X_MIN,
37          ROTATE_KNEE_LEFT_X_MAX,
38          0,
39          0,
40          0,
41          0));
```

Die Methode `addJoint(Joint j)` in der Klasse `Configuration` fügt der Liste der Gelenke das übergebene hinzu. Die Methode `updateEndEffector()` und `getEndeffektor()` bestimmen die Koordinaten des Endeffektors. Die Methode `getJointCount()` zählt die Anzahl der Gelenke in der Liste und mit der Methode `getJoint(int index)` bekommt man das Gelenk an der Stelle index in der Liste zurückgegeben.

5.3 Die Klasse Computation

Die Methode `compute()` in der Klasse `Computation` enthält die eigentliche Berechnung der inversen Kinematik. Es gibt 3 Abbruchbedingungen: die Berechnung wird abgebrochen, wenn der Zielpunkt bis zu einem bestimmten Grenzwert (ε) erreicht ist. Es wird die Distanz zwischen dem Endeffektor und dem Zielpunkt berechnet. Wenn diese kleiner als ε ist, dann werden die Werte aus den TransformGroups in der Kopie in das Original übertragen. Geprüft wird nach jeder Bewegung des Endeffektors.

Außerdem bricht die Berechnung ab, wenn eine bestimmte Anzahl an Versuchen gemacht wurde, das Ziel zu erreichen. Auch dann werden die aktuellen Werte der TransformGroups der Kopie in das Original übertragen. Diese Versuche werde in der äußersten Schleife der Berechnung gezählt.

Und die Berechnung bricht ab, wenn die Gelenke nicht mehr bewegt werden. Mit einer bool'schen Variable wird geprüft, ob wenigstens ein Gelenk bewegt wurde. Wenn dies nicht der Fall ist, wird die Hauptschleife abgebrochen und die Winkel werden in die Originalmarionette übertragen.

Dann gibt es noch 2 innere Schleifen. Die erste der beiden geht die Gelenke von hinten nach vorne durch. Da es auch Gelenke gibt, die mehr als eine Drehachse (die Arme beispielsweise) haben, überprüft die zweite innere Schleife, um welche Achsen in dem aktuellen Gelenk gedreht werden darf.

Ein Problem war, dass es im Dreidimensionalen 3 Drehebenen gibt. Wenn beispielsweise der Arm der Marionette um x gedreht werden soll, darf dies nur geschehen, wenn eine Drehung um x erlaubt ist. Bei meiner ersten Version der inversen Kinematik Simulation wurde jedes Gelenk um den errechneten Winkel gedreht. Das führte zu dem Problem, dass die Marionette Bewegungen machte, die gar nicht erlaubt waren. Also mußte sichergestellt werden, dass das Gelenk nur gedreht wird, wenn die Achse auch erlaubt ist. Abbildung 17 zeigt die Lösung zu dem Problem. G ist der Zielpunkt den der Endeffktor erreichen soll. Dieser liegt allerdings nicht in der Drehebene der Achse a. Der Winkel zwischen dem Endeffektor E und dem Zielpunkt G, der berechnet werden würde, wäre falsch. Also muß man den Zielpunkt in die Drehebene projezieren und dann den Winkel zwischen dem Endeffektor E und dem Zielpunkt G' berechnen. Das gleiche geschieht auch mit dem Endeffektor, falls er noch nicht in der Ebene liegt. Mathematisch gesehen ist es eine einfache Schnittpunktberechnung zwischen einer Geraden und einer Ebene.

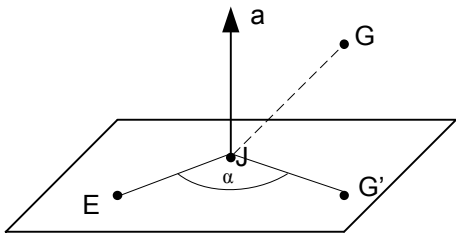


Abbildung 17: Drehebene

Wichtig ist eine Abfrage, ob der Cosinus von a größer gleich 1 ist. Cosinus gleich 1 heißt, dass der Drehwinkel 0 ist, also keine Drehung gemacht werden soll. Der Endeffektor und der Zielpunkt liegen schon auf einer Linie. Man bricht dann also den aktuellen Schleifendurchlauf ab und geht zum nächsten über.

Am Ende der Berechnung muss noch überprüft werden, in welche Richtung gedreht werden muss. In Kapitel 2.3 wurde schon erwähnt, dass man im Dreidimensionalen eine andere Überprüfung machen muß, als im Zweidimensionalen. Alledings ist nur eine kleine Erweiterung zu oben genannter Methode nötig. Man berechnet das Kreuzprodukt zwischen den normierten Vektoren vom Gelenkmittelpunkt zu der in die Drehebene projezierten

Endeffektor- bzw. Zielposition. Man erhält einen Vektor der senkrecht auf den beiden steht. Dann berechnet man das Skalarprodukt zwischen diesem neuen Vektor und der Drehachse. Wenn das Skalarprodukt größer als 0 ist, dann rotiert man gegen den Uhrzeigersinn und anderenfalls im Uhrzeigersinn. Diese Rotationen werden dann in die betroffenen TransformGroups der Kopie geschrieben.

Die Klasse `Computation` hat noch eine weitere Methode, `getAngle (Transform3D rotation, int axis)`. Diese gibt den aktuellen Drehwinkel der angegebenen TransformGroup zurück.

Im Anhang findet man den gesamten Code der beschriebenen Klassen.

6 Steuerung der Marionette

Die Steuerung der Marionette erfolgt durch eine Simulation eines Marionettenkreuzes. Das Kreuz, an dem Marionette hängt, sieht aus wie in Abbildung 18 gezeigt. Man kann das Kreuz um x, y und z drehen. Die Fäden sind im Mittelpunkt der Unterarme bzw. der Unterschenkel befestigt. Somit befindet sich auch der Endeffektor an dieser Stelle. Die Fäden sind dabei im Prinzip unendlich lang, jede Bewegung des Kreuzes verursacht eine relative Bewegung jedes Fadens. Diese Bewegung wird einfach an den jeweiligen Endeffektor weitergegeben.

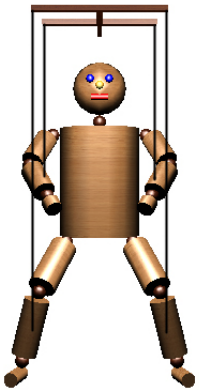


Abbildung 18: Marionettenkreuz

Abbildung 19 zeigt wie man sich die Drehung des Kreuzes vorstellen muss. Die Abbildung zeigt beispielhaft eine Drehung der Marionette um die z-Achse. Das Kreuz wird durch die Cursortasten bewegt. Mit jedem Druck wird das Kreuz um einen bestimmten Winkel α gedreht. Die Ruhestellung des Kreuzes ist parallel zur x-Achse. Außerdem ist die Drehung des Kreuzes beschränkt auf ein Maximum und ein Minimum. Nach jeder Bewegung des Kreuzes muß nun eine neue Zielposition, also die neue x-, y- und z-Position des Kreuzes und somit auch des Endeffektors, bestimmt werden.

In der Klasse `Marionette` gibt es die Methode `rotateCross()`, in der die Simulation des Kreuzes implementiert ist. Die Methode bekommt einen Winkel übergeben und die Achse, um die gedreht werden soll. Um später die Veränderung jeder Endeffektorposition bestimmen zu können, muß man sich die alten Winkelstellungen des Kreuzes merken. Aus den neuen Winkelstel-

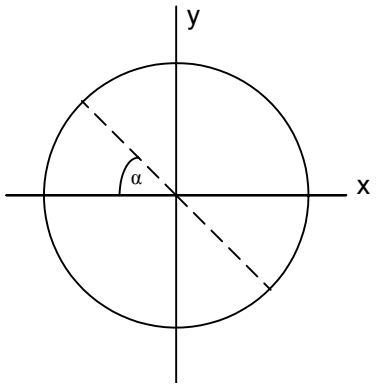


Abbildung 19: Drehung des Marionettenkreuzes um die z-Achse

lungen des Kreuzes ergeben sich die neuen Positionen der Aufhängpunkte der Fäden. Bildet man nun die Differenzen aus der jeweiligen neuen und der alten Position, erhält man ein Delta, welches auf den entsprechenden Effektor addiert wird, um so den neuen Zielpunkt zu erhalten.

Zusätzlich kann man die Marionette noch drehen, hoch und runter sowie nach rechts und links bewegen. Das hat aber nichts mit inverser Kinematik zu tun.

Im Anhang findet man den Code der oben beschriebenen Methode.

7 Fazit und Ausblick

Das Ziel der Arbeit war es, in einem bestehenden Rendering System (Java3D) eine Marionette zu implementieren, die sich mit Hilfe inverser Kinematik realistisch bewegen lässt. Das Ergebnis sieht man in Tabelle 1 und 2. Die Bilder zeigen was passiert, wenn man das Marionettenkreuz um die X bzw. Z-Achse dreht.

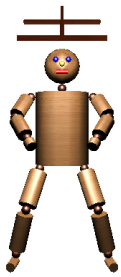
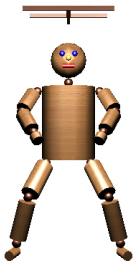
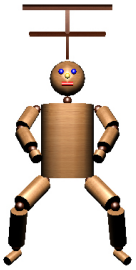
		
Cursortaste Runter: Marionettenkreuz dreht sich gegen den Uhrzeigersinn um die X-Achse	Marionette in Ruhestellung	Cursortaste Hoch: Marionettenkreuz dreht sich im Uhrzeigersinn um die X-Achse

Tabelle 1: Drehen des Marionettenkreuzes um die X-Achse

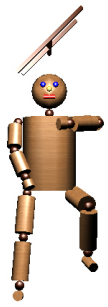
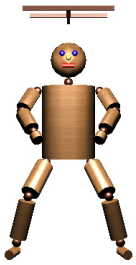
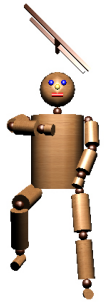
		
Cursortaste Links: Marionettenkreuz dreht sich gegen den Uhrzeigersinn um die Z-Achse	Marionette in Ruhestellung	Cursortaste Rechts: Marionettenkreuz dreht sich im Uhrzeigersinn um die Z-Achse

Tabelle 2: Drehen des Marionettenkreuzes um die Z-Achse

Die größte Hürde stellte erwartungsgemäß die Umsetzung der inversen Kinematik dar, da sämtliche Literatur lediglich Vorgehensweisen und Implementationen für den zweidimensionalen Fall enthält. Zu den Schwierigkeiten und Besonderheiten des dreidimensionalen Falls siehe Abschnitt 5.

Am Ende möchte ich noch kurz erwähnen, was man noch verbessern kann:

- Für die Bewegung von Kopf und Hände und Füße sind alle TransformGroups vorhanden. Man kann die Steuerung noch so verbessern, dass man den Kopf, die Hände und die Füße auch noch bewegen kann. Weiterhin sollte es noch eine Funktion geben, mit der man an den einzelnen Fäden der Marionette ziehen kann und sich dadurch Arme und Beine unahängig voneinander bewegen lassen.

- Möglich ist die Konstruktion eines realen Holzkreuzes mit Trackingsensoren, mit dessen Hilfe sich die Bewegungen der Marionette steuern lassen.

- Optimierung der Darstellung der Marionette hinsichtlich Texturen, Geometrien sowie Beleuchtung.

- Simulation von Schwerkraft, Kollisionserkennung und Aufzeichnung und Wiedergabe von Bewegungsabläufen.

Anhang

Listing 1: Joint.java

```
1  public class Joint {
2
3    public static final int AXIS_X = 0;
4    public static final int AXIS_Y = 1;
5    public static final int AXIS_Z = 2;
6
7    public String name = null;
8
9    private double rotationXMin = 0;
10   private double rotationXMax = 0;
11   private double rotationYMin = 0;
12   private double rotationYMax = 0;
13   private double rotationZMin = 0;
14   private double rotationZMax = 0;
15
16   private TransformGroup tgRotationCloneX = null;
17   private TransformGroup tgRotationCloneY = null;
18   private TransformGroup tgRotationCloneZ = null;
19   private TransformGroup tgRotationOriginalX = null;
20   private TransformGroup tgRotationOriginalY = null;
21   private TransformGroup tgRotationOriginalZ = null;
22
23   private Node referenceNode = null;
24
25   public Joint(
26     String name,
27     TransformGroup tgRotationOriginalX,
28     TransformGroup tgRotationOriginalY,
29     TransformGroup tgRotationOriginalZ,
30     BranchGroup clone,
31     double rotationXMin,
32     double rotationXMax,
33     double rotationYMin,
34     double rotationYMax,
35     double rotationZMin,
36     double rotationZMax) {
37     this.name = name;
38     this.tgRotationOriginalX = tgRotationOriginalX;
39     this.tgRotationOriginalY = tgRotationOriginalY;
40     this.tgRotationOriginalZ = tgRotationOriginalZ;
41     tgRotationCloneX =
```

```
42      (tgRotationOriginalX != null)
43        ? Marionette.findJoint(
44          clone,
45          (String) (tgRotationOriginalX.getUserData()))
46        : null;
47    tgRotationCloneY =
48      (tgRotationOriginalY != null)
49        ? Marionette.findJoint(
50          clone,
51          (String) (tgRotationOriginalY.getUserData()))
52        : null;
53    tgRotationCloneZ =
54      (tgRotationOriginalZ != null)
55        ? Marionette.findJoint(
56          clone,
57          (String) (tgRotationOriginalZ.getUserData()))
58        : null;
59    this.rotationXMin = Math.toRadians(rotationXMin);
60    this.rotationXMax = Math.toRadians(rotationXMax);
61    this.rotationYMin = Math.toRadians(rotationYMin);
62    this.rotationYMax = Math.toRadians(rotationYMax);
63    this.rotationZMin = Math.toRadians(rotationZMin);
64    this.rotationZMax = Math.toRadians(rotationZMax);
65    if (tgRotationCloneX != null)
66      referenceNode = tgRotationCloneX;
67    else if (tgRotationCloneY != null)
68      referenceNode = tgRotationCloneY;
69    else if (tgRotationCloneZ != null)
70      referenceNode = tgRotationCloneZ;
71  }
72
73  public Point3d getWorldCoordinates() {
74    Transform3D tfPosition =
75      Marionette.getWorldCoordinates(referenceNode);
76    Point3d position = new Point3d(0, 0, 0);
77    tfPosition.transform(position);
78    return position;
79  }
80
81  public boolean isAxisSupported(int axis) {
82    switch (axis) {
83      case AXIS_X :
84        return (tgRotationCloneX != null);
85      case AXIS_Y :
86        return (tgRotationCloneY != null);
```

```
87        case AXIS_Z :
88          return (tgRotationCloneZ != null);
89        default :
90          return false;
91      }
92    }
93
94    public TransformGroup getTransformGroup(
95      int axis,
96      boolean original) {
97      switch (axis) {
98        case AXIS_X :
99          return original
100           ? tgRotationOriginalX
101           : tgRotationCloneX;
102       case AXIS_Y :
103         return original
104           ? tgRotationOriginalY
105           : tgRotationCloneY;
106       case AXIS_Z :
107         return original
108           ? tgRotationOriginalZ
109           : tgRotationCloneZ;
110       default :
111         return null;
112     }
113   }
114
115   public double getMin(int axis) {
116     switch (axis) {
117       case AXIS_X :
118         return rotationXMin;
119       case AXIS_Y :
120         return rotationYMin;
121       case AXIS_Z :
122         return rotationZMin;
123       default :
124         return 0;
125     }
126   }
127
128   public double getMax(int axis) {
129     switch (axis) {
130       case AXIS_X :
131         return rotationXMax;
```

```
132       case AXIS_Y :
133         return rotationYMax;
134       case AXIS_Z :
135         return rotationZMax;
136       default :
137         return 0;
138     }
139   }
140 }
```

Listing 2: Configuration.java

```
1  public class Configuration {
2
3    private Vector joints = null;
4    private Node endEffectorNodeClone = null;
5    private Point3d endEffector = null;
6
7    public Configuration(
8      Node endEffectorNodeOriginal,
9      BranchGroup clone) {
10     joints = new Vector();
11     endEffector = new Point3d();
12     endEffectorNodeClone =
13       Marionette.findNode(
14         clone,
15         (String)endEffectorNodeOriginal.getUserData());
16   }
17
18   public void addJoint(Joint j) {
19     joints.add(j);
20   }
21
22   private void updateEndEffector() {
23     endEffector.set(0, 0, 0);
24     if (endEffectorNodeClone == null)
25       return;
26     Transform3D dummy =
27       Marionette.getWorldCoordinates(endEffectorNodeClone);
28     dummy.transform(endEffector);
29   }
30
31   public int jointCount() {
32     return joints.size();
```

```
  }

  public Joint getJoint(int index) {
    if (index >= joints.size() || index < 0)
      return null;
    return (Joint)joints.elementAt(index);
  }

  public Point3d getEndEffector() {
    updateEndEffector();
    return endEffector;
  }

}
```

Listing 3: Computation.java

```
public class Computation {

  public static final double DAMPING = 0.01;
  public static final int MAX_TRIES = 10;
  public static final double EPSILON = 0.005;

  private Configuration configuration;
  private Vector3d goal;

  public Computation(
    Configuration configuration,
    Vector3d goal) {
    this.configuration = configuration;
    this.goal = goal;
  }

  private double getAngle(Transform3D rotation, int axis) {
    Matrix3d m = new Matrix3d();
    rotation.getRotationScale(m);

    if (axis == Joint.AXIS_X) {
      return Math.atan2(m.m21, m.m11);
    } else if (axis == Joint.AXIS_Y) {
      return Math.atan2(m.m02, m.m00);
    } else {
      return Math.atan2(m.m10, m.m11);
    }
```

```
28  }
29
30  public void compute() {
31    for (int tries = 0; tries < MAX_TRIES; tries++) {
32      boolean atLeastOneSuccessful = false;
33      for (int i = configuration.jointCount() - 1;
34        i >= 0;
35        i--) {
36        Joint currentJoint = configuration.getJoint(i);
37
38        for (int axis = Joint.AXIS_X;
39          axis <= Joint.AXIS_Z;
40          axis++) {
41          if (!currentJoint.isAxisSupported(axis))
42            continue;
43          Vector3d rotationAxis = new Vector3d();
44          switch (axis) {
45            case Joint.AXIS_X :
46              {
47                rotationAxis.set(1, 0, 0);
48                break;
49              }
50            case Joint.AXIS_Y :
51              {
52                rotationAxis.set(0, 1, 0);
53                break;
54              }
55            case Joint.AXIS_Z :
56              {
57                rotationAxis.set(0, 0, 1);
58                break;
59              }
60          }
61          // Endeffektor in die Ebene holen
62          Vector3d jointWorld =
63            new Vector3d(
64              currentJoint.getWorldCoordinates());
65          Vector3d currentEnd =
66            new Vector3d(configuration.getEndEffector());
67          double lamdaEnd =
68            jointWorld.dot(rotationAxis)
69              - (rotationAxis.dot(currentEnd));
70          Vector3d lamdaVecEnd = new Vector3d(rotationAxis);
71          lamdaVecEnd.scale(lamdaEnd);
72          Vector3d newCurrentEnd = new Vector3d(currentEnd);
```

```
73            newCurrentEnd.add(lamdaVecEnd);
74
75            // Zielpunkt in die Ebene holen
76            Vector3d desiredEnd = new Vector3d(goal);
77            double lamdaGoal =
78              jointWorld.dot(rotationAxis)
79                - rotationAxis.dot(desiredEnd);
80            Vector3d lamdaVecGoal =
81              new Vector3d(rotationAxis);
82            lamdaVecGoal.scale(lamdaGoal);
83            Vector3d newDesiredEnd = new Vector3d(desiredEnd);
84            newDesiredEnd.add(lamdaVecGoal);
85
86            // Vector vom aktuellen Gelenk zum neuen Endeffektor
87            newCurrentEnd.sub(jointWorld);
88            // Vector vom aktuellen Gelenk zum Zielpunkt
89            newDesiredEnd.sub(jointWorld);
90            // Normalisieren
91            Vector3d currentEndNorm =
92              new Vector3d(newCurrentEnd);
93            currentEndNorm.normalize();
94            Vector3d desiredEndNorm =
95              new Vector3d(newDesiredEnd);
96            desiredEndNorm.normalize();
97            // Winkel zwischen currentEnd' und desiredEnd'
98            double cosinus =
99              desiredEndNorm.dot(currentEndNorm);
100           // cosinus = 1 bedeutet keine drehung
101           if (cosinus > 0.9999999)
102             continue;
103
104           atLeastOneSuccessful = true;
105
106           // Test in welche Richtung gedreht wird
107           Vector3d cross = new Vector3d();
108           cross.cross(newCurrentEnd, newDesiredEnd);
109           double direction = cross.dot(rotationAxis);
110           Transform3D tfDummy = new Transform3D();
111           currentJoint.getTransformGroup(
112             axis,
113             false).getTransform(
114             tfDummy);
115
116           double desiredAngleRel = Math.acos(cosinus);
117           double currentAngleAbs = getAngle(tfDummy, axis);
```

```
118
119             if (direction > 0) {
120               // CCW, rotieren gegen Uhrzeigersinn
121               double desiredAngleAbs =
122                 Math.min(
123                   currentJoint.getMax(axis),
124                   Math.max(
125                     currentAngleAbs + desiredAngleRel,
126                     currentJoint.getMin(axis)));
127               switch (axis) {
128                 case Joint.AXIS_X :
129                   {
130                     tfDummy.rotX(
131                       Math.min(
132                         desiredAngleAbs,
133                         currentAngleAbs + DAMPING));
134                     break;
135                   }
136                 case Joint.AXIS_Y :
137                   {
138                     tfDummy.rotY(
139                       Math.min(
140                         desiredAngleAbs,
141                         currentAngleAbs + DAMPING));
142                     break;
143                   }
144                 case Joint.AXIS_Z :
145                   {
146                     tfDummy.rotZ(
147                       Math.min(
148                         desiredAngleAbs,
149                         currentAngleAbs + DAMPING));
150                     break;
151                   }
152               }
153               currentJoint.getTransformGroup(
154                 axis,
155                 false).setTransform(
156                 tfDummy);
157             } else {
158               // CW, rotieren im Uhrzeigersinn
159               double desiredAngleAbs =
160                 Math.min(
161                   currentJoint.getMax(axis),
162                   Math.max(
```

```
163                 currentAngleAbs - desiredAngleRel,
164                 currentJoint.getMin(axis)));
165             switch (axis) {
166               case Joint.AXIS_X :
167                 {
168                   tfDummy.rotX(
169                     Math.max(
170                       desiredAngleAbs,
171                       currentAngleAbs - DAMPING));
172                   break;
173                 }
174               case Joint.AXIS_Y :
175                 {
176                   tfDummy.rotY(
177                     Math.max(
178                       desiredAngleAbs,
179                       currentAngleAbs - DAMPING));
180                   break;
181                 }
182               case Joint.AXIS_Z :
183                 {
184                   tfDummy.rotZ(
185                     Math.max(
186                       desiredAngleAbs,
187                       currentAngleAbs - DAMPING));
188                   break;
189                 }
190             }
191             currentJoint.getTransformGroup(
192               axis,
193               false).setTransform(
194               tfDummy);
195           }
196
197           // Test ob Zielgebiet ereicht
198           Vector3d distance =
199             new Vector3d(configuration.getEndEffector());
200           distance.sub(desiredEnd);
201           if (distance.length() < EPSILON) {
202             for (int j = configuration.jointCount() - 1;
203               j >= 0;
204               j--) {
205               currentJoint = configuration.getJoint(j);
206               for (int a = Joint.AXIS_X;
207                 a <= Joint.AXIS_Z;
```

```
208                 a++) {
209                 if (!currentJoint.isAxisSupported(a))
210                   continue;
211                 currentJoint.getTransformGroup(
212                   a,
213                   false).getTransform(
214                   tfDummy);
215                 currentJoint.getTransformGroup(
216                   a,
217                   true).setTransform(
218                   tfDummy);
219               }
220             }
221             return;
222           }
223         }
224       }
225       if (!atLeastOneSuccessful)
226         break;
227     }
228     Transform3D tfDummy = new Transform3D();
229     for (int i = configuration.jointCount() - 1;
230       i >= 0;
231       i--) {
232       Joint currentJoint = configuration.getJoint(i);
233       for (int a = Joint.AXIS_X; a <= Joint.AXIS_Z; a++) {
234         if (!currentJoint.isAxisSupported(a))
235           continue;
236         currentJoint.getTransformGroup(
237           a,
238           false).getTransform(
239           tfDummy);
240         currentJoint.getTransformGroup(
241           a,
242           true).setTransform(
243           tfDummy);
244       }
245     }
246   }
247 }
```

Listing 4: Methode rotateCross() aus Marionette.java

```
1   public void rotateCross(double angle, int axis) {
```

```
2     double oldAngleX = crossAngleX;
3     double oldAngleY = crossAngleY;
4     double oldAngleZ = crossAngleZ;
5     if (axis == AXIS_X)
6       crossAngleX =
7         clamp(
8           crossAngleX + Math.toRadians(angle),
9           Math.toRadians(CROSS_ROTATION_X_MIN),
10          Math.toRadians(CROSS_ROTATION_X_MAX));
11    else if (axis == AXIS_Y)
12      crossAngleY =
13        clamp(
14          crossAngleY + Math.toRadians(angle),
15          Math.toRadians(CROSS_ROTATION_Y_MIN),
16          Math.toRadians(CROSS_ROTATION_Y_MAX));
17    else
18      crossAngleZ =
19        clamp(
20          crossAngleZ + Math.toRadians(angle),
21          Math.toRadians(CROSS_ROTATION_Z_MIN),
22          Math.toRadians(CROSS_ROTATION_Z_MAX));
23
24    rotate.rotX(crossAngleX);
25    tgRotateCrossX.setTransform(rotate);
26    rotate.rotY(crossAngleY);
27    tgRotateCrossY.setTransform(rotate);
28    rotate.rotZ(crossAngleZ);
29    tgRotateCrossZ.setTransform(rotate);
30
31    Matrix3d mRotX = new Matrix3d();
32    mRotX.rotX(oldAngleX);
33    Matrix3d mRotY = new Matrix3d();
34    mRotY.rotY(oldAngleY);
35    Matrix3d mRotZ = new Matrix3d();
36    mRotZ.rotZ(oldAngleZ);
37
38    // Faeden Arm alt
39
40    Point3d pCrossXArm = new Point3d(crossLengthXArm, 0, 0);
41    Point3d pOldResultArm = new Point3d();
42
43    Matrix3d dummyMatrix = new Matrix3d(mRotX);
44    dummyMatrix.mul(mRotY);
45    dummyMatrix.mul(mRotZ);
46    dummyMatrix.transform(pCrossXArm, pOldResultArm);
```

```
47
48      // Faeden Bein alt
49
50      Point3d pCrossZLeg =
51        new Point3d(crossLengthXLeg, 0, crossLengthZLeg);
52      Point3d pOldResultLeg = new Point3d();
53
54      dummyMatrix.transform(pCrossZLeg, pOldResultLeg);
55
56      // Faeden Arm neu
57
58      Point3d pResultArm = new Point3d();
59
60      mRotX.rotX(crossAngleX);
61      mRotY.rotY(crossAngleY);
62      mRotZ.rotZ(crossAngleZ);
63
64      dummyMatrix.set(mRotX);
65      dummyMatrix.mul(mRotY);
66      dummyMatrix.mul(mRotZ);
67
68      dummyMatrix.transform(pCrossXArm, pResultArm);
69
70      pResultArm.sub(pOldResultArm);
71
72      // Faeden Bein neu
73
74      Point3d pResultLeg = new Point3d();
75
76      dummyMatrix.transform(pCrossZLeg, pResultLeg);
77
78      pResultLeg.sub(pOldResultLeg);
79
80      // Linker Arm
81
82      Vector3d endEffectorPosArmLeft =
83        new Vector3d(configurationArmLeft.getEndEffector());
84      switch (axis) {
85        case AXIS_X :
86          {
87            endEffectorPosArmLeft.add(pResultArm);
88            break;
89          }
90        case AXIS_Y :
91          {
```

```
92            endEffectorPosArmLeft.add(pResultArm);
93            break;
94          }
95        case AXIS_Z :
96          {
97            endEffectorPosArmLeft.add(pResultArm);
98            break;
99          }
100     }
101     Computation computeArmLeft =
102       new Computation(
103         configurationArmLeft,
104         endEffectorPosArmLeft);
105     computeArmLeft.compute();
106
107     // Rechter Arm
108
109     Vector3d endEffectorPosArmRight =
110       new Vector3d(configurationArmRight.getEndEffector());
111     switch (axis) {
112       case AXIS_X :
113         {
114           endEffectorPosArmRight.add(pResultArm);
115           break;
116         }
117       case AXIS_Y :
118         {
119           endEffectorPosArmRight.sub(pResultArm);
120           break;
121         }
122       case AXIS_Z :
123         {
124           endEffectorPosArmRight.sub(pResultArm);
125           break;
126         }
127     }
128     Computation computeArmRight =
129       new Computation(
130         configurationArmRight,
131         endEffectorPosArmRight);
132     computeArmRight.compute();
133
134     // linkes Bein
135
136     Vector3d endEffectorPosLegLeft =
```

```
137       new Vector3d(configurationLegLeft.getEndEffector());
138     switch (axis) {
139       case AXIS_X :
140         {
141           endEffectorPosLegLeft.add(pResultLeg);
142           break;
143         }
144       case AXIS_Y :
145         {
146           endEffectorPosLegLeft.add(pResultLeg);
147           break;
148         }
149       case AXIS_Z :
150         {
151           endEffectorPosLegLeft.add(pResultLeg);
152           break;
153         }
154     }
155     Computation computeLegLeft =
156       new Computation(
157         configurationLegLeft,
158         endEffectorPosLegLeft);
159     computeLegLeft.compute();
160 
161     // rechtes Bein
162 
163     Vector3d endEffectorPosLegRight =
164       new Vector3d(configurationLegRight.getEndEffector());
165     switch (axis) {
166       case AXIS_X :
167         {
168           endEffectorPosLegRight.add(pResultLeg);
169           break;
170         }
171       case AXIS_Y :
172         {
173           endEffectorPosLegRight.sub(pResultLeg);
174           break;
175         }
176       case AXIS_Z :
177         {
178           endEffectorPosLegRight.sub(pResultLeg);
179           break;
180         }
181     }
```

```
182     Computation computeLegRight =
183       new Computation(
184         configurationLegRight,
185         endEffectorPosLegRight);
186     computeLegRight.compute();
187 
188   }
```

Literatur

[1] Dennis J. Bouvier. *The Java3D Tutorial*. http://java.sun.com/products/java-media/3D/collateral/, 2002.

[2] Jeff Lander. *Making Kine More Flexible*. http://www.darwin3d.com/gamedev/articles/col1198.pdf, 1998.

[3] Rick Parent. *Computer Animation, Algorithms and Techniques*. 2002.