

Computergrafik II

Bericht

Andreas Heinze
Matr.-Nr. 1015157

27. Juli 2007

Inhaltsverzeichnis

1	Einleitung	4
1.1	Animation	4
1.2	Kinematik	4
1.3	Starrkörpersysteme	4
2	Gelenkarmroboter	6
2.1	Übung	6
2.2	Lösungsansatz	6
2.2.1	Offene kinematische Ketten	6
2.2.2	Vorwärtskinematik	7
2.2.3	Freiheitsgrad	7
2.2.4	Umsetzung in OpenGL	7
2.3	Darstellung	8
2.4	Quellcode	11
3	ebenes Gelenkviereck	15
3.1	Übung	15
3.2	Lösungsansatz	16
3.2.1	Nullstelleniteration	16
3.2.2	Schnitt am Gelenkpaar	17
3.2.3	Körperschnitt	17
3.3	Darstellung	17
3.4	Quellcode	20
4	Räumliches Gelenkviereck	32
4.1	Übung	32
4.2	Lösungsansatz	33
4.2.1	Starrkörper	33
4.2.2	Aufbau	33
4.2.3	Berechnung	33
4.2.4	Bedienung	34
4.3	Darstellung	34
4.4	Quellcode	36

1 Einleitung

Dies ist der Abschlußbericht zur Lehrveranstaltung “Computergrafik II”, gelesen von Herrn Prof. Dr.-Ing. René Klingenberg. Thema dieses Berichts sind **offene und geschlossene kinematische Ketten** und deren Programmierung mit *OpenGL*.

Quelle für diesen Bericht ist das Skript zur Lehrveranstaltung sowie die Website des Österreichers Karlheinz Zeitner <http://www.zeiner.at/c/index.html>. Die Programme sind von mir selbst programmiert. Einzig der Quelltext aus Kapitel 3 stammt von Johannes Feldmann und wurde mit seiner Zustimmung in diesem Bericht verwendet.

Jedes Kapitel beginnt mit der Erläuterung von Begriffen und/oder Rechenschritten, setzt sich fort mit Darstellungen zur bildlichen Erklärung und schließt ab mit dem vollständigen Quellcode.

1.1 Animation

Mit Animation bezeichnet man in der Computergrafik **bewegte Bilder, die in ihrer Gesamtheit einen definierten Ablauf rechnergestützt darstellen**. Die Darstellung wird durch das Anzeigen einer Folge von computergenerierten Bildern in einem festen oder variablen Zeitabstand ermöglicht. Um Bewegungen und Objekte möglichst naturgetreu wiederzugeben, werden bei Animationen verschiedene Indikatoren wie z.B. *Schattierung, Materialbeschaffenheit und -eigenschaften* und physikalische Größen mit einbezogen.

1.2 Kinematik

Die Kinematik befasst sich mit dem **geometrischen Ablauf der Bewegung von Körpern und Punkten im Raum**, unabhängig von den Bewegungsursachen und -folgen. Wichtige Größen sind *Weg (Bahn), Geschwindigkeit* und *Beschleunigung*. Kinematik ist als Teilgebiet der theoretischen Mechanik ein Bestandteil der Animation und für die physikalisch korrekte Darstellung von Bewegungen unverzichtbar.

1.3 Starrkörpersysteme

Die folgenden Kapitel befassen sich mit der Animation der Lage und Lageveränderungen von Starrkörpern, die durch Gelenke gemeinsam zu einer **kinematischen Kette** verbunden sind. Durch Lageveränderungen der Gelenke G_n innerhalb ihrer Freiheiten f_n wird Bewegung in der Kette erzeugt.

Am Beispiel eines Gelenkarmroboters werden in Kapitel 2 mathematische Eigenschaften der hierarchischen Anordnung (**offene kinematische Kette**) erläutert. In Kapitel 3 wird die ge-

schlossene kinematische Kette im zweidimensionalen Raum (*Ebene*) vorgestellt, in Kapitel 4 die **geschlossene kinematische Kette** im dreidimensionalen Raum.

2 Gelenkarmroboter

2.1 Übung

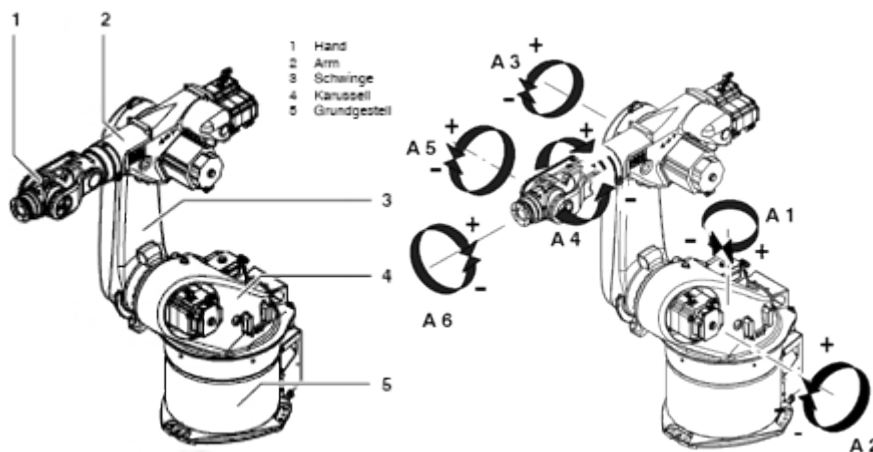


Abbildung 2.1: Hauptbestandteile, Drehachsen und Drehsinn beim Verfahren eines Gelenkarmroboters am Beispiel KUKA KR 30-3/KR 60-3/KR 30 L16

In der ersten Übung geht es um die Konstruktion eines Gelenkarmroboters im dreidimensionalen Raum, dessen Bewegungseigenschaften das Prinzip der **offenen kinematischen Kette in der Animation** näherbringen. Die Starrkörper werden hier nur als einfache gerade Strecken im Raum dargestellt, weil in dieser Übung die korrekte Datenaufbereitung und die Kombination der erforderlichen Transformationen zur Simulation von Drehachsen und Drehsinn im Vordergrund stehen. Als Vorgabe gilt der industrielle Gelenkroboter KR 30-3 der Kuka Roboter GmbH, wie in Abb. 2.1 dargestellt.

Der gesamte Roboter ist zu animieren. Für die Berechnungen wird C bzw. C++ verwendet, für die Graphik OpenGL.

2.2 Lösungsansatz

2.2.1 Offene kinematische Ketten

Die Anzahl der Starrkörper ist gleich der Anzahl der verbindenden Gelenke ($n_K = n_G$). Durch diese Gegebenheit ist keine Schleife innerhalb des Starrkörpersystems möglich, also liegt eine **hierarchische Anordnung** vor, innerhalb welcher der Weg zwischen zwei Starrkörpern *stets eindeutig* ist. Da die Starrkörper je nach Bedarf verästelt angeordnet sein können, spricht dabei man auch von einer **Baumstruktur**.

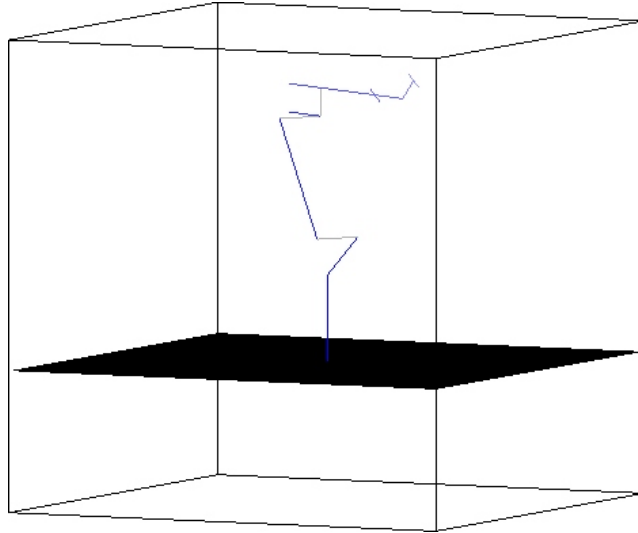


Abbildung 2.2: Nachbildung des Gelenkarmroboters mit geraden Strecken als Starrkörper

2.2.2 Vorwärtskinematik

Als Bezugssystem gilt das Koordinatensystem K_0 , das als **Ausgangspunkt für die Bestimmung der Lage bis zum äußersten Körper** dient. In Analogie erhält das erste Gelenk die Bezeichnung G_0 .

Jeder weitere in die Hierarchie eingeordnete Körper ist in seiner Lage abhängig von den Lagen der vorhergehenden Körper und Gelenkkoordinaten. Deshalb wird in der Vorwärtskinematik die Lage jedes Körpers in Abhängigkeit zur Lage und Form seines in der Hierarchie übergeordneten Vorgängers und der Transformationen des dazwischen liegenden Gelenks errechnet.

2.2.3 Freiheitsgrad

Der Freiheitsgrad f eines Gelenks G_n wird in Form einer Verlagerung des Ausgangskordinatensystems $K_{n'}$ gegenüber dem Eingangskordinatensystem K_n durch **drei Translationen und drei Rotationen**, die voneinander abhängig sind, beschrieben. Somit ergeben sich sechs Koordinaten, die den Gelenkfreiheitsgrad $f_G = 6 - n_b$ bestimmen, wobei n_b die Anzahl der geometrischen Bindungen, also der mathematischen Beziehungen, bezeichnet.

Im Programm wird auf die Errechnung des Freiheitsgrades des hier vorliegenden Starrkörpersystems (Abb. 2.2) $f = 6n_K - \sum_{i=1}^{n_K} n_{b_i} = 6n_K - \sum_{i=1}^{n_K} (6 - f_{G_i})$ verzichtet, weil der Wert in diesem Beispiel für die Darstellung nicht erforderlich ist.

2.2.4 Umsetzung in OpenGL

Das Programm ist in C++ geschrieben und bedient sich der Funktionen aus den OpenGL-Bibliotheken von FreeGLUT. In einer Endlosschleife, die nicht per Tastendruck beendet werden kann, wird der Roboter als Linienmodell in einem **Drahtkäfig** auf einer schwarzen Ebene dargestellt. Je tiefer sich ein Körper in der Hierarchie befindet, desto heller ist sein Farbton. Der Drahtkäfig mit dem darin enthaltenen Roboter lässt sich bei gedrückter Taste per Mausbewegung um seine X- und Z-Achse drehen. Die dafür verwendeten Variablen heißen `mouseAngleX`

und `mouseAngleZ`.

Das Prinzip der Vorwärtskinematik lässt sich sehr einfach durch **Aneinanderreihung der Transformationen und Darstellungen** in `glPushMatrix-/glPopMatrix`-Konstrukten mit OpenGL realisieren. Da einige Körper des Gelenkroboters versetzt angebracht sind, werden sie auch im Modell versetzt dargestellt. Dieser Versatz wird durch graue Verbindungslinien als Verbindungsbolzen wiedergegeben.

Sowohl die Maße der Körper als auch die Bewegungsfreiräume der Gelenke werden in angemessenem Verhältnis wiedergegeben. Die Rotationswinkel der n Gelenke werden in einer Variablen `rotationAngle1-n` gespeichert. Sie lassen sich in 10-Grad-Abstufung wie folgt über Tasten steuern:

Tastencode	Gelenk	Achse	Richtung
1	G_1	Y	aufsteigend
2	G_1	Y	absteigend
3	G_2	Z	aufsteigend
4	G_2	Z	absteigend
5	G_3	Z	aufsteigend
6	G_3	Z	absteigend
7	G_4	X	aufsteigend
8	G_4	X	absteigend
9	G_5	X	aufsteigend
0	G_5	X	absteigend
q	G_6	Y	aufsteigend
w	G_6	Y	absteigend

Im Folgenden ist eine Bewegung aller Gelenke G_{1-6} des Starrkörpersystems dargestellt. Die Werte der Gelenke werden in jedem Schritt um 10 Grad dekrementiert.

2.3 Darstellung

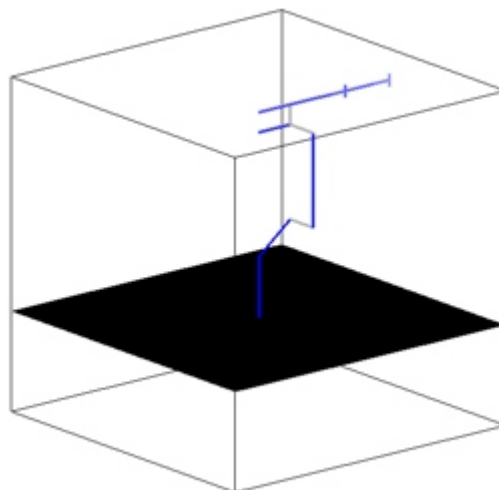


Abbildung 2.3: $G_1=0$, $G_2=90$, $G_3=-90$, $G_4=0$, $G_5=0$, $G_6=0$

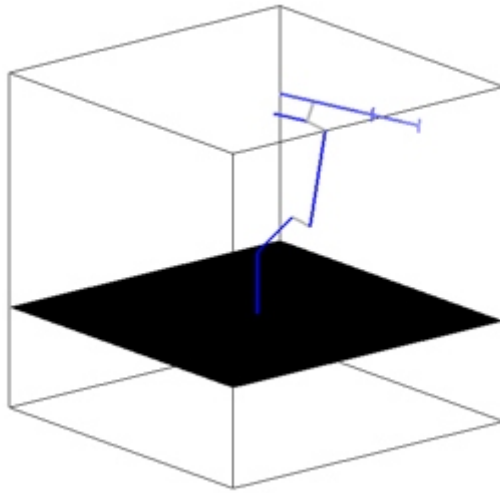


Abbildung 2.4: $G_1=-10$, $G_2=80$, $G_3=-100$, $G_4=-10$, $G_5=-10$, $G_6=-10$

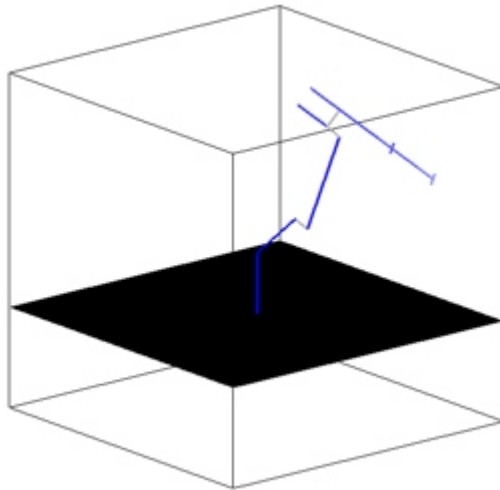


Abbildung 2.5: $G_1=-10$, $G_2=70$, $G_3=-110$, $G_4=-20$, $G_5=-20$, $G_6=-20$

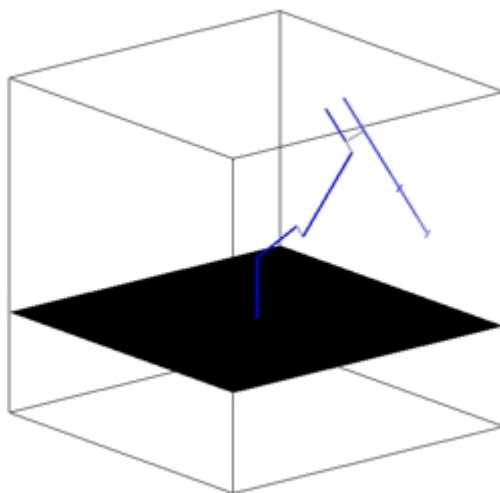


Abbildung 2.6: $G_1=-10$, $G_2=60$, $G_3=-120$, $G_4=-30$, $G_5=-30$, $G_6=-30$

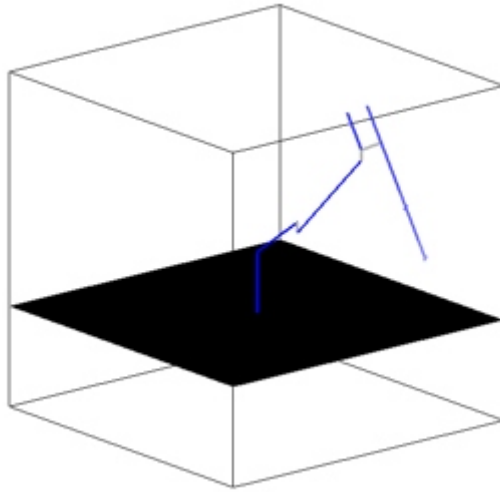


Abbildung 2.7: $G_1=-10$, $G_2=50$, $G_3=-130$, $G_4=-40$, $G_5=-40$, $G_6=-40$

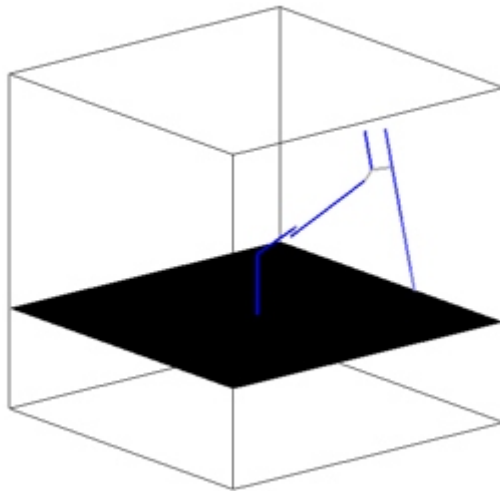


Abbildung 2.8: $G_1=-10$, $G_2=40$, $G_3=-140$, $G_4=-50$, $G_5=-50$, $G_6=-50$

2.4 Quellcode

```
1 #include <GL/gl.h>
2 #include <GL/glu.h>
3 #include <GL/glut.h>
4 #include <cstdio>
5 #include <cmath>
6
7 void display();
8 void motion(int x, int y);
9 void passiveMotion(int x, int y);
10 void keyboard(unsigned char key, int x, int y);
11 void reshape (int w, int h);
12
13 float rotationAngle1 = 0.0;
14 float rotationAngle2 = 90.0;
15 float rotationAngle3 = -90.0;
16 float rotationAngle4 = 0.0;
17 float rotationAngle5 = 0.0;
18 float rotationAngle6 = 0.0;
19
20 double xOld, yOld;
21
22 GLfloat mouseAngleX = 0.0;
23 GLfloat mouseAngleZ = 0.0;
24
25 int main( int argc, char** argv )
26 {
27     // pass command line arguments to GLUT (if any)
28     glutInit( &argc, argv );
29     glutInitDisplayMode (GLUT_SINGLE | GLUT_RGB);
30     // create 700 x 700 display
31     glutInitWindowSize( 700, 700 );
32     glutCreateWindow( "Roboterarm" );
33     // register GL display function
34     glutDisplayFunc( display );
35     glutReshapeFunc(reshape);
36     glutKeyboardFunc( keyboard );
37     glutMotionFunc(motion);
38     glutPassiveMotionFunc(passiveMotion);
39     // enter GLUT main loop
40     glutMainLoop();
41
42     // this point will never be reached
43     return 0;
44 }
45
46 /*
47  * GL display function
48  */
49 void display()
50 {
51     // clear display
52     glClear( GL_COLOR_BUFFER_BIT );
```

```

53  glClearColor(1.0, 1.0, 1.0, 0.0);
54  glColor3f(0.0, 0.0, 0.0);
55  glPushMatrix();
56      glRotatef(mouseAngleX, 1.0, 0.0, 0.0);
57      glRotatef(mouseAngleZ, 0.0, 1.0, 0.0);
58
59  glPushMatrix();
60      glTranslatef(0.0, 0.2, 0.0);
61      glutWireCube(1.0);
62  glPopMatrix();
63
64  glPushMatrix();
65
66  glBegin(GL_QUADS);
67      glVertex3f(-0.5, 0.0, 0.5); glVertex3f(0.5, 0.0, 0.5); glVertex3f(0.5, 0.0, -0.5); glVertex3f(-0.5, 0.0,
        -0.5); // Boden
68  glEnd();
69
70  glColor3f (0.0, 0.0, 1.0);
71  glRotatef (rotationAngle1, 0.0, 1.0, 0.0);
72  glBegin( GL_LINES );
73      glVertex3f( 0.0, 0.0, 0.0 ); glVertex3f( 0.0, 0.181, 0.0 );
74      glVertex3f( 0.0, 0.181, 0.0 ); glVertex3f( 0.117, 0.2717, 0.0 );
75  glEnd();
76
77  glColor3f (0.7, 0.7, 0.7); // Verbindung zur Illustration des Verbindungsbolzen, welcher die versetzten
        Elemente verbindet
78  glBegin( GL_LINES );
79      glVertex3f( 0.117, 0.2717, 0.1036); glVertex3f( 0.117, 0.2717, 0.0);
80  glEnd();
81
82  glTranslatef (0.117, 0.2717, 0.0);
83  glRotatef( rotationAngle2, 0.0, 0.0, 1.0);
84  glColor3f (0.1, 0.1, 1.0);
85  glBegin( GL_LINES );
86      glVertex3f( 0.0, 0.0, 0.1036 ); glVertex3f( 0.283, 0.0, 0.1036);
87  glEnd();
88
89  glColor3f (0.7, 0.7, 0.7); // Verbindung zur Illustration des Verbindungsbolzen, welcher die versetzten
        Elemente verbindet
90  glBegin( GL_LINES );
91      glVertex3f( 0.283, 0.0, 0.1036); glVertex3f( 0.283, 0.0, 0.0);
92  glEnd();
93
94  glTranslatef (0.283, 0.0, 0.0);
95  glRotatef (rotationAngle3, 0.0, 0.0, 1.0);
96  glColor3f (0.2, 0.2, 1.0);
97  glBegin (GL_LINES);
98      glVertex3f(-0.117, 0.0, 0.0); glVertex3f (0.0, 0.0, 0.0);
99  glEnd ();
100
101  glColor3f (0.7, 0.7, 0.7); // Verbindung zur Illustration des Verbindungsbolzen, welcher die versetzten
        Elemente verbindet
102  glBegin( GL_LINES );

```

```

103     glVertex3f( 0.0, 0.0, 0.0); glVertex3f( 0.0, 0.06, 0.0);
104 glEnd();
105
106 glTranslatef (0.0, 0.06, 0.0);
107 glRotatef (rotationAngle4, 1.0, 0.0, 0.0);
108 glColor3f (0.3, 0.3, 1.0);
109 glBegin (GL_LINES);
110     glVertex3f(-0.117, 0.0, 0.0); glVertex3f (0.155, 0.0, 0.0);
111 glEnd ();
112
113 glTranslatef (0.155, 0.0, 0.0);
114 glRotatef (rotationAngle5, 1.0, 0.0, 0.0);
115 glColor3f (0.4, 0.4, 1.0);
116 glBegin (GL_LINES);
117     glVertex3f(0.0, 0.0, 0.0); glVertex3f (0.155, 0.0, 0.0);
118     glVertex3f(0.05, -0.02, 0.0); glVertex3f (0.05, 0.02, 0.0); // Querer Balken zur Illustration der
119     Drehung
120 glEnd ();
121
122 glTranslatef (0.155, 0.0, 0.0);
123 glRotatef (rotationAngle6, 0.0, 1.0, 0.0);
124 glColor3f (0.5, 0.5, 1.0);
125 glBegin (GL_LINES);
126     glVertex3f(0.0, 0.0, 0.0); glVertex3f (0.057, 0.0, 0.0);
127 glEnd ();
128
129 glTranslatef (0.057, 0.0, 0.0);
130 glColor3f (0.6, 0.6, 1.0);
131 glBegin (GL_LINES);
132     glVertex3f(0.0, -0.02, 0.0); glVertex3f (0.0, 0.02, 0.0);
133 glEnd ();
134
135 glPopMatrix();
136 glPopMatrix();
137 // display everything
138 glFlush();
139 }
140 void keyboard( unsigned char key, int x, int y)
141 {
142     switch (key)
143     {
144         case '1': if (rotationAngle1 < 185) rotationAngle1 +=10.0; break;
145         case '2': if (rotationAngle1 > -185) rotationAngle1 -=10.0; break;
146         case '3': if (rotationAngle2 < 135) rotationAngle2 +=10.0; break;
147         case '4': if (rotationAngle2 > -35) rotationAngle2 -=10.0; break;
148         case '5': if (rotationAngle3 < 158) rotationAngle3 +=10.0; break;
149         case '6': if (rotationAngle3 > -120) rotationAngle3 -=10.0; break;
150         case '7': if (rotationAngle4 < 350) rotationAngle4 +=10.0; break;
151         case '8': if (rotationAngle4 > -350) rotationAngle4 -=10.0; break;
152         case '9': if (rotationAngle5 < 119) rotationAngle5 +=10.0; break;
153         case '0': if (rotationAngle5 > -119) rotationAngle5 -=10.0; break;
154         case 'q': rotationAngle6 +=10.0; break;
155         case 'w': rotationAngle6 -=10.0; break;

```

```
156     }
157
158     glutPostRedisplay();
159 }
160
161 void reshape (int w, int h)
162 {
163     glViewport (0, 0, (GLsizei) w, (GLsizei) h);
164     glMatrixMode (GL_PROJECTION);
165     glLoadIdentity ();
166 }
167
168 void motion(int x, int y)
169 {
170     mouseAngleX = fmod(mouseAngleX + 0.3 * (y - yOld) + 1800.0, 360.0);
171     mouseAngleZ = fmod(mouseAngleZ + 0.3 * (x - xOld) + 1800.0, 360.0);
172     xOld = x;
173     yOld = y;
174     glutPostRedisplay();
175 }
176
177 void passiveMotion(int x, int y)
178 {
179     xOld = x;
180     yOld = y;
181 }
```

3 ebenes Gelenkviereck

3.1 Übung

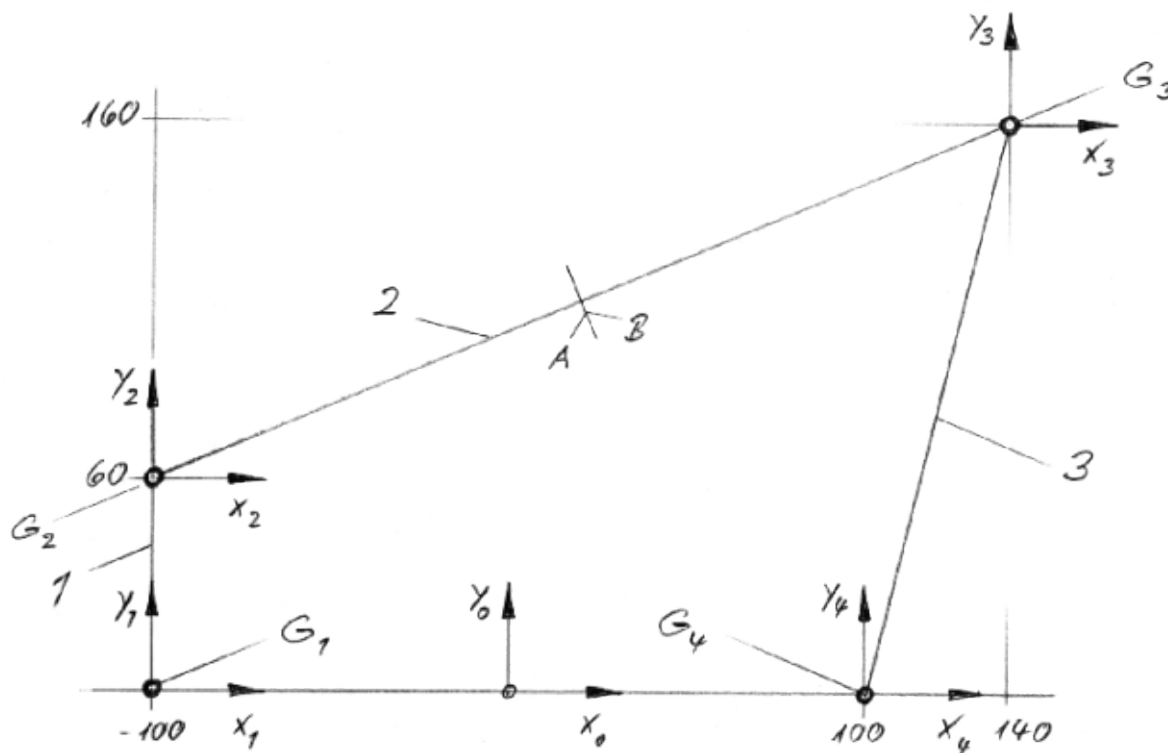


Abbildung 3.1: Geometrie eines ebenen Gelenkvierecks in der Ausgangslage (Maßzahlen in K_0)

In dieser Aufgabe sind die möglichen geometrischen Lagen des ebenen Gelenkvierecks aus Abbildung 3.1 graphisch darzustellen.

Um zu einem akzeptablen Ergebnis zu kommen, wird die Vorwärtskinematik genutzt. Zunächst wird die geschlossene kinematische Kette in zwei offene kinematische Teilketten (links und rechts) aufgeteilt.

Dann wird sich im Normalfall der Nullstellenbestimmung einer transzendenten Gleichung bzw. eines transzendenten Gleichungssystems bedient, allerdings bietet sich wegen der Komplexität das einfachere Newton-Raphson-Verfahren als numerische Alternative an.

Im ersten Lösungsansatz soll eine transzendente Gleichung (Schnitt am Gelenkpaar) iteriert werden. Dadurch wird die Bewegung des Gelenkvierecks unter Einbeziehung des Winkels θ_4 animiert.

Der zweite Lösungsansatz umfasst die Iteration des Systems aus drei transzendenten Gleichun-

gen im Form eines Körperschnitts. Im Gegensatz zum ersten Ansatz werden hier auch die Gelenkwinkel θ_2 und θ_3 ermittelt.

3.2 Lösungsansatz

Um eine Bewegung zu erzeugen, wird der Winkel θ_1 verändert und in dessen Abhängigkeit auch die übrigen Winkel. Die Schrittweite von θ_1 ist im Programm auf 5 festgelegt. Die Schrittweite der übrigen Winkel ergibt sich durch die Ermittlung jedes Winkels mittels Nullstelleniteration. Da die Gelenke durch Starrkörper miteinander verbunden sind, ändern sich die Distanzen zwischen benachbarten Gelenken nicht. Somit sind auch ihre Koordinatensysteme K_i stets in konstantem Abstand zu ihren benachbarten Koordinatensystemen K_{i-1} und K_{i+1} .

3.2.1 Nullstelleniteration

Das Newtonverfahren hilft beim Suchen einer Nullstelle mittels Iteration. Dazu wird eine Schleife höchstens n -mal durchlaufen, bis eine bestimmte Winkeldifferenz (im Programm definiert durch die Variable `epsilon`) erreicht wird. n ist frei wählbar und garantiert äquivalent zu seiner Höhe einen genauen Wert. Das Verfahren wird in der Funktion `double newton(double t1, double t2)` realisiert und umgesetzt:

```
1 // Nullstelle nach Newtonverfahren rekursiv bestimmen
2 double newton(double t1, double t2){
3     int iterationstiefe = 0;
4     double t2Alt = 0;
5     double winkeldifferenz = 0;
6
7     do{
8         iterationstiefe++;
9         double ableitung = ( g1((t2 + delta), t1) - g1(t2, t1) ) / delta;
10
11         t2Alt = t2;
12         t2 = t2Alt - ( g1(t2, t1)/ableitung );
13         winkeldifferenz = fabs(fabs(t2) - fabs(t2Alt)); // fabs() liefert Absolutwert
14
15     }while( (winkeldifferenz > epsilon) && (iterationstiefe < 20) );
16
17     return t2;
18 }
```

3.2.2 Schnitt am Gelenkpaar

Die Schließbedingungen zur Gewinnung des ersten Gleichungssystems werden anders aufgestellt als beim Körper- oder Gelenkschnitt. Sie werden in der Funktion `double g1(double t4, double x)` umgesetzt, hier beim Schnitt am Gelenkpaar (G_2, G_3).

```
1 // aus CG-II-Skript, Abschnitt 2.3.2.3 Schnitt am Gelenkpaar
2 double g1(double t4, double x)
3 {
4     return (sqrt( ( (200 + 40* cos(t4) + 60* sin(x) - 160* sin(t4)) *
5                   (200 + 40* cos(t4) + 60* sin(x) - 160* sin(t4)) )
6                   +
7                   ( (-60* cos(x) + 160* cos(t4) + 40* sin(t4)) *
8                   (-60* cos(x) + 160* cos(t4) + 40* sin(t4)) )
9                   ))
10    - 260; // 260 = Schliessbedingung = Laenge der Koppel
11 }
```

3.2.3 Körperschnitt

Während man beim Schnitt am Gelenkpaar nur die Koordinate θ_4 aus nur einer transzendenten Gleichung gewinnt, benötigt man beim Körperschnitt 3 transzendenten Gleichungen, um 3 abhängige Koordinaten zu ermitteln. Als unerlässlich gilt die Jacobi-Matrix, die beim Iterationsvorgang als Abbruchkriterium gilt, wenn sie zu Null wird.

Der in Kapitel 3.4 folgende Quellcode von Johannes Feldmann macht den Lösungsweg in den Funktionen `double g1(double t1, double t2, double t3, double t4)`, `double g2(double t1, double t2, double t3, double t4)` und `double g3(double t1, double t2, double t3, double t4)` (die drei Gleichungen), `void calcJacobianMatrix()` (Berechnung der Jacobi-Matrix) und `void calcAngles()` (Iterationsschleife) deutlich.

3.3 Darstellung

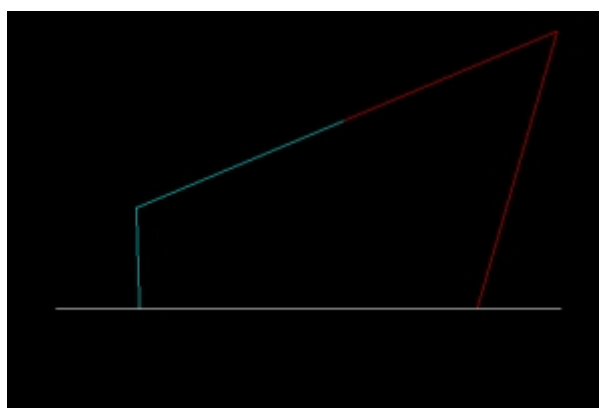


Abbildung 3.2: ebenes Gelenkviereck, Figur 1

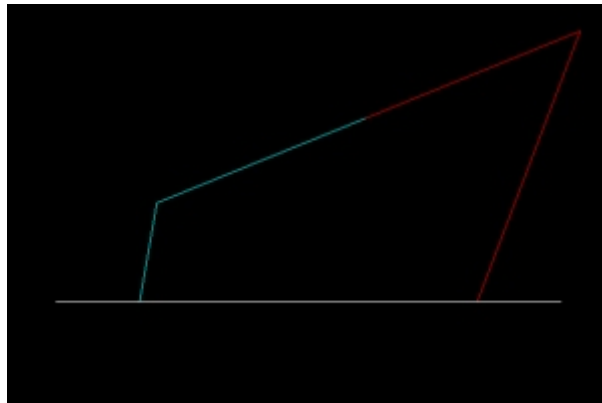


Abbildung 3.3: ebenes Gelenkviereck, Figur 2

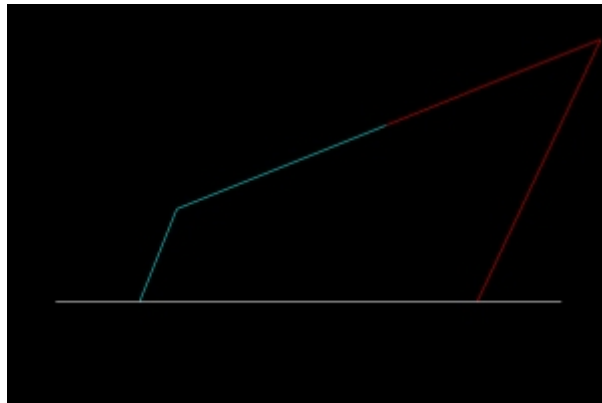


Abbildung 3.4: ebenes Gelenkviereck, Figur 3

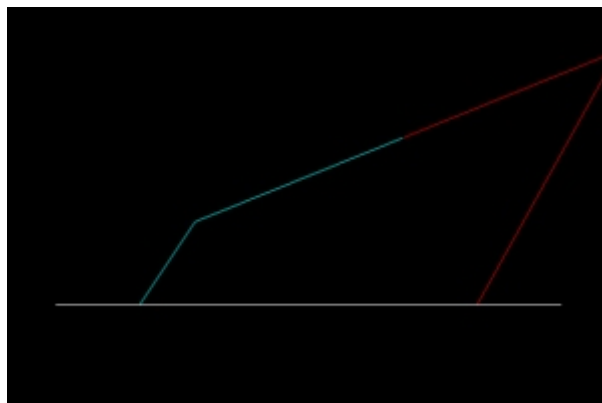


Abbildung 3.5: ebenes Gelenkviereck, Figur 4

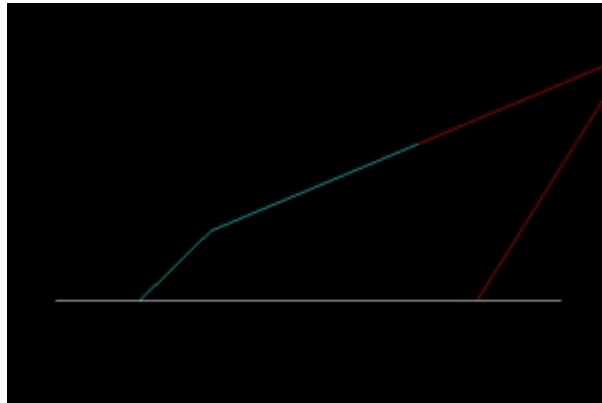


Abbildung 3.6: ebenes Gelenkviereck, Figur 5

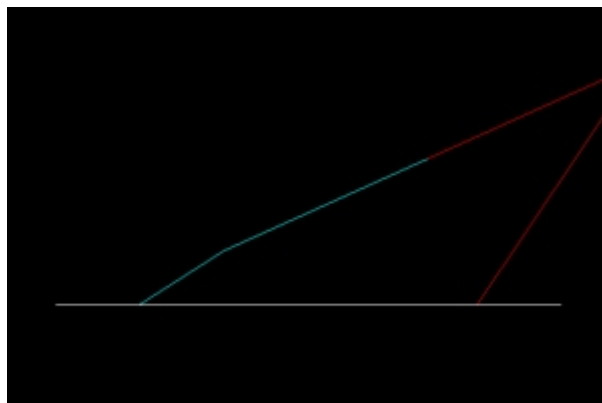


Abbildung 3.7: ebenes Gelenkviereck, Figur 6

3.4 Quellcode

```
1  /*
2     Computergrafik II
3     FH Hannover / SS 2007
4     Johannes Feldmann (1015254)
5
6     Uebung 3.2
7     Koeperschnitt an kinematisch geschlossener Kette
8
9     Steuerung links/rechts mit Cursortasten
10 */
11
12 #include<GL/gl.h>
13 #include<GL/glu.h>
14 #include<GL/glut.h>
15 #include<iostream>
16 #include<cmath>
17 #include"matrix.h"
18
19 using namespace std;
20
21 FILE *fout = stdout;
22
23 // angles
24 double theta1 = 0.0;
25 double theta2 = 0.0;
26 double theta3 = 0.0;
27 double theta4 = 0.0;
28 double theta2_tmp = 0.0;
29 double theta3_tmp = 0.0;
30 double theta4_tmp = 0.0;
31
32 double fkn_vect[3]; /*Funktionsvertevektor*/
33 double erg_vect[3]; /*Ergebnisvektor der Matrixmultiplikation*/
34 double epsilon = 0.00001; /*Abbruchkriterium fuer Iteration*/
35 double delta = 0.000001; /*Argumentwertaenderung fuer Differenzenquotient*/
36 double angle_diff[3];
37 double jacobianMatrix[3][6];
38 double inverseJacobianMatrix[3][6];
39
40 float lookZ = 390.0f;
41
42 int iterations = 0;
43 int theta_index=0;
44
45 // declarations
46 void init();
47 void display();
48 void reshape(int w, int h);
49 void specialKey(int key, int x, int y );
50 void calcAngles(); // berechnet Winkel
51 void initGlut();
52
```

```

53
54 /*Gleichung 1 aus Skript*/
55 double g1(double t1, double t2, double t3 , double t4){
56     return ( -200 -40*cos(t4)- 60*sin(t1)
57             +50*( -cos(t2)*sin(t1) - cos(t1)*sin(t2) )
58             +120*( cos(t1)*cos(t2) - sin(t1)*sin(t2) )
59             +160*sin(t4)
60             +50*( -cos(t4)*sin(t3) - cos(t3)*sin(t4) )
61             +120*( cos(t3)*cos(t4) - sin(t3)*sin(t4) ));
62 }
63
64 /*Gleichung 2 aus Skript*/
65 double g2(double t1, double t2, double t3 , double t4){
66     return 60*cos(t1)-160*cos(t4)
67           +120*( cos(t2)*sin(t1) + cos(t1)*sin(t2) )
68           +50*( cos(t1)*cos(t2) - sin(t1)*sin(t2) )
69           -40*sin(t4)
70           +120*( cos(t4)*sin(t3) + cos(t3)*sin(t4) )
71           +50*( cos(t3)*cos(t4) - sin(t3)*sin(t4) );
72 }
73
74 /*Gleichung 3 aus skript*/
75 double g3(double t1, double t2, double t3 , double t4){
76     return cos(t1)*cos(t2) - cos(t3)*cos(t4) - sin(t1)*sin(t2)
77           +sin(t3)*sin(t4);
78 }
79
80
81 /*
82     Berechnet die Jakobimatrix
83     Auf Basis der Lsung von Kommilitone Hannes Burmeister aus dem SS 2006
84 */
85 void calcJacobianMatrix(){
86     fkn_vect[0] = g1(theta1, theta2, theta3, theta4);
87     fkn_vect[1] = g2(theta1, theta2, theta3, theta4);
88     fkn_vect[2] = g3(theta1, theta2, theta3, theta4);
89
90     #ifdef DEBUG_MAIN
91         cout<< " fkn_vect[0]:." <<fkn_vect[0]<<endl;
92         cout<< " fkn_vect[1]:." << fkn_vect[1]<<endl;
93         cout<< " fkn_vect[2]:." << fkn_vect[2]<<endl;
94     #endif
95
96     /*g1 - g3 partiell nach theta2 abgeleitet (Sekante)*/
97     jacobianMatrix[0][0] = (g1(theta1, theta2+delta, theta3, theta4) - fkn_vect[0])/ (delta);
98     jacobianMatrix[1][0] = (g2(theta1, theta2+delta, theta3, theta4) - fkn_vect[1])/ (delta);
99     jacobianMatrix[2][0] = (g3(theta1, theta2+delta, theta3, theta4) - fkn_vect[2])/ (delta);
100
101     jacobianMatrix[0][1] = (g1(theta1, theta2, theta3+delta, theta4) - fkn_vect[0])/ (delta);
102     jacobianMatrix[1][1] = (g2(theta1, theta2, theta3+delta, theta4) - fkn_vect[1])/ (delta);
103     jacobianMatrix[2][1] = (g3(theta1, theta2, theta3+delta, theta4) - fkn_vect[2])/ (delta);
104
105     jacobianMatrix[0][2] = (g1(theta1, theta2, theta3, theta4+delta) - fkn_vect[0])/ (delta);
106     jacobianMatrix[1][2] = (g2(theta1, theta2, theta3, theta4+delta) - fkn_vect[1])/ (delta);

```

```

107     jacobianMatrix[2][2] = (g3(theta1, theta2, theta3, theta4+delta) - fkn_vect[2]) / (delta);
108 }
109
110
111 /*
112     Berechnet theta 2-4 in Abhngigkeit von theta1
113     Auf Basis der Lsung von Kommilitone Hannes Burmeister aus dem SS 2006
114 */
115 void calcAngles(){
116     iterations=0;
117     do{
118         iterations += 1;
119
120         #ifdef DEBUG_MAIN
121             cout<<endl<<" Iterationsdurchlauf:_" <<iterations<<endl;
122         #endif
123
124         calcJacobianMatrix();
125         Inverse(jacobianMatrix, inverseJacobianMatrix, 3);
126
127         for(int i=0; i<3; i++){
128             erg_vect[i] =0.0;
129             for(int j=0; j<3;j++){
130                 erg_vect[i] += inverseJacobianMatrix[i][j]*fkn_vect[j];
131             }
132         }
133
134         theta2_tmp = theta2;
135         theta3_tmp = theta3;
136         theta4_tmp = theta4;
137
138         theta2 = theta2_tmp - erg_vect[0];
139         theta3 = theta3_tmp - erg_vect[1];
140         theta4 = theta4_tmp - erg_vect[2];
141
142         angle_diff[0] = fabs(theta2_tmp) - fabs(theta2);
143         angle_diff[1] = fabs(theta3_tmp) - fabs(theta3);
144         angle_diff[2] = fabs(theta4_tmp) - fabs(theta4);
145
146     }while( ( fabs(angle_diff[0]) > epsilon &&
147             fabs(angle_diff[1]) > epsilon &&
148             fabs(angle_diff[2]) > epsilon )
149           || iterations > 10 );
150
151 }
152
153
154 int main (int argn, char **argc)
155 {
156     glutInit( &argn, argc );
157     initGlut();
158     calcAngles(); // Winkel fuer theta1=0 berechnen
159     glutMainLoop();
160

```

```

161     return 0;
162 }
163
164
165 void init()
166 {
167     glEnable(GL_DEPTH_TEST);
168     glClearDepth(1.0);
169     glDepthFunc(GL_LESS);
170     // black background color
171     glClearColor( 0, 0, 0, 1.0 );
172 }
173
174
175 void initGlut() {
176     glutInitDisplayMode( GLUT_RGB | GLUT_DEPTH | GLUT_DOUBLE);
177     glutInitWindowSize( 700, 700 );
178     glutCreateWindow( " Geschlossene_kinematische_Kette_-_Koerperschnitt" );
179     glutSpecialFunc( specialKey );
180     glutReshapeFunc( reshape );
181     glutDisplayFunc( display );
182 }
183
184
185 void reshape(int w, int h)
186 {
187     float aspect = static_cast<float>( w ) / static_cast<float>( h );
188     // set projection transformation
189     glMatrixMode( GL_PROJECTION );
190     glLoadIdentity();
191     //glOrtho( -300.0, 300.0 , -300.0, 300.0, -300.0, 300.0);
192
193     gluPerspective( 55.0, aspect, 100.0, 800.0 );
194     glMatrixMode( GL_MODELVIEW );
195     glLoadIdentity();
196     //gluLookAt( 0.0, 5.0, lookZ, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0 );
197
198     // set viewport
199     glViewport( 0, 0, w, h );
200 }
201
202
203 /*
204  * GL special key function
205  */
206 void specialKey( int key, int x, int y )
207 {
208     switch( key )
209     {
210         case GLUT_KEY_LEFT:
211             theta1 += 0.02;
212
213             if(theta_index == 62) theta_index = 0;
214             else theta_index += 1;

```

```

215
216     if(theta1 > 2*M_PI){
217         theta1 = 0.0;
218         theta2 = 0.0;
219         theta3 = 0.0;
220         theta4 = 0.0;
221     }
222     break;
223
224     case GLUT_KEY_RIGHT:
225         theta1 -= 0.02;
226
227         if(theta_index == 0) theta_index = 62;
228         else theta_index -= 1 ;
229
230         if(theta1 < -2*M_PI){
231             theta1 = 0.0;
232             theta2 = 0.0;
233             theta3 = 0.0;
234             theta4 = 0.0;
235         }
236         break;
237     }
238
239     calcAngles();
240     glutPostRedisplay();
241 }
242
243 /*
244     GLUT display – Funktion, zeichnet Gelenkkper
245 */
246 void display(){
247
248     #ifdef DEBUG_MAIN
249         cout<<"theta1:_"<<theta1*(180.0/M_PI)<<endl;
250         cout<<"theta2:_"<<theta2*(180.0/M_PI)<<endl;
251         cout<<"theta3:_"<<theta3*(180.0/M_PI)<<endl;
252         cout<<"theta4:_"<<theta4*(180.0/M_PI)<<endl;
253         cout<<"+++++"<<endl;
254     #endif
255     /*
256         cout<<"+++++"<<endl;
257         cout << theta_index << endl;
258         cout<<"theta1: "<<theta1*(180.0/M_PI)<<endl;
259         cout<<"theta2: "<<theta2*(180.0/M_PI)<<endl;
260         cout<<"theta3: "<<theta3*(180.0/M_PI)<<endl;
261         cout<<"theta4: "<<theta4*(180.0/M_PI)<<endl;
262     */
263
264     glClear( GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
265     glMatrixMode(GL_MODELVIEW);
266

```

```

267     glLoadIdentity();
268     gluLookAt( 0, 100.0, lookZ, 0, 50.0, 0, 0.0, 1.0, 0.0 );
269
270     glPushMatrix();
271
272         glColor3f(1,1,1); // white
273         glBegin(GL_LINES);
274             glVertex3f(-150.0, 0, 0);
275         glVertex3f(150.0, 0, 0);
276         glEnd();
277
278         glColor3f(0,1,1);
279
280
281         /*linke Teilkette*/
282         glPushMatrix();
283         glTranslatef(-100.0, 0.0, 0.0 );
284         glRotatef( theta1*(180.0/M_PI), 0.0, 0.0, 1.0 );
285
286         /*Koerper 1 zeichnen*/
287         glBegin(GL_LINES);
288             glVertex3f(0.0, 0.0, 0.0);
289             glVertex3f(0.0, 60.0, 0);
290         glEnd();
291
292         glTranslatef(0.0, 60.0, 0.0 );
293         glRotatef( theta2*(180.0/M_PI), 0.0, 0.0, 1.0 );
294
295         /*koerper 2.1 zeichnen*/
296         glBegin(GL_LINES);
297             glVertex3f(0.0, 0.0, 0.0);
298             glVertex3f(120.0, 50.0, 0.0 );
299         glEnd();
300     glPopMatrix();
301
302     glColor3f(1,0,0);
303
304     /*rechte Teilkette*/
305     glPushMatrix();
306         glTranslatef(100.0, 0.0, 0.0 );
307         glRotatef( theta4*(180.0/M_PI), 0.0, 0.0, 1.0 );
308
309         /*Koerper 4 zeichnen*/
310         glBegin(GL_LINES);
311             glVertex3f(0.0, 0.0, 0.0);
312             glVertex3f(40.0, 160.0, 0);
313         glEnd();
314
315         glTranslatef(40.0, 160.0, 0.0 );
316         glRotatef( theta3*(180.0/M_PI), 0.0, 0.0, 1.0 );
317
318         /*koerper 2.2 zeichnen*/
319         glBegin(GL_LINES);
320

```

```

321         glVertex3f(0.0, 0.0, 0.0);
322         glVertex3f(-120.0, -50.0, 0.0 );
323     glEnd();
324
325     glPopMatrix();
326
327     glPopMatrix();
328
329     glutSwapBuffers(); // let glut draw everything
330 }

```

matrix.h:

```

1 // File Matrix.h
2 #ifndef __MATRIX_H__
3 #define __MATRIX_H__
4
5 #include <stdio>
6
7 #define DEBUG 1
8 #define NMAX 6
9
10
11 int Gaussalg (double a[][NMAX], int n, double x[]);
12 int Inverse (double a[][NMAX], double ainv[][NMAX], int n);
13 void MatIn (FILE *f, double a[][NMAX], int *n, int *m);
14 void MatOut (FILE *f, double a[][NMAX], int n, int m);
15 void Produkt (double a[][NMAX], double b[][NMAX], double c[][NMAX], int n);
16 void Copy(double a[][NMAX], double b[][NMAX], int n);
17
18
19 #endif

```

matrix.c:

```

1 // Matrix.c
2 // Modul mit einigen Matrix-Methoden
3
4 #include <math.h>
5 #include <stdio.h>
6 #include "Matrix.h"
7
8 // Kopiert die quadratische n x n Matrix a und speichert sie in b
9 // b = a
10 void Copy(double a[][NMAX], double b[][NMAX], int n)
11 {
12     int i, j;
13     for (i = 0; i < n; i++)
14         for (j = 0; j < n; j++)
15             b[i][j] = a[i][j];
16 }
17
18 // Gauss'sches Eliminationsverfahren mit Zeilenpivotisierung
19 // Argumente:

```

```

20 // double a[N][N+1] erweiterte Koeffizientenmatrix Read/Write
21 // int n Anzahl der Gleichungen Read
22 // double x[N] Loesungen Write
23 // Resultat:
24 // int Fehlercode 0 fuer Fehler, 1 fuer Erfolg
25
26 int Gaussalg (double a[][NMAX], int n, double x[])
27 {
28     int i, j; // Zeile, Spalte
29     int s; // Eliminationsschritt
30     int pzeile; // Pivotzeile
31     int fehler = 0; // Fehlerflag
32     double f; // Multiplikationsfaktor
33     const double Epsilon = 0.01; // Genauigkeit
34     double Maximum; // Zeilenpivotisierung
35     extern FILE *fout;
36
37     s = 0;
38     do { // die einzelnen Eliminationsschritte
39         //fprintf(fout, "Schritt %2i von %2i\n", s+1, n-1);
40         Maximum = fabs(a[s][s]); // groesstes Element
41         pzeile = s; // suchen
42         for (i = s+1; i < n; i++)
43             if (fabs(a[i][s]) > Maximum) {
44                 Maximum = fabs(a[i][s]);
45                 pzeile = i;
46             }
47         fehler = (Maximum < Epsilon);
48         if (fehler) break; // nicht loesbar
49
50         if (pzeile != s) // falls erforderlich, Zeilen tauschen
51             { double h;
52                 for (j = s; j <= n; j++) {
53                     h = a[s][j];
54                     a[s][j] = a[pzeile][j];
55                     a[pzeile][j] = h;
56                 }
57             }
58
59         // Elimination --> Nullen in Spalte s ab Zeile s+1
60         for (i = s + 1; i < n; i++) {
61             f = -(a[i][s]/a[s][s]); // Multiplikationsfaktor
62             a[i][s] = 0.0;
63             for (j = s+1; j <= n; j++) // die einzelnen Spalten
64                 a[i][j] += f*a[s][j]; // Addition der Zeilen i, s
65         }
66         #if DEBUG
67             MatOut (stdout, a, n, n+1);
68         #endif
69         s++;
70     } while ( s < n-1 );
71
72     if (fehler)
73     {

```

```

74     fprintf (fout, "gauss:_Gleichungssystem_nicht_loesbar\n");
75     return 0;
76 }
77 else
78 {
79     // Berechnen der Loesungen aus der entstandenen Dreiecksmatrix
80     // letzte Zeile
81     x[n-1] = a[n-1][n] / a[n-1][n-1];
82     // restliche Zeilen
83     for (i = n-2 ; i >= 0; i-- )
84     {
85         for (j = n-1 ; j > i ; j-- )
86         {
87             a[i][n] -= x[j]*a[i][j]; // rechte Seite berechnen
88         }
89         x[i] = a[i][n] / a[i][i]; // Loesung
90     }
91     return 1;
92 }
93 }
94 }
95
96 // Inverse einer Matrix
97 // Berechnung nach dem Gauss–Jordan Verfahren
98 // Lit. Helmut Selder, Einfhruung in die numerische Mathematik fr Ingenieure, HANSER
99 //
100 // Das Verfahren Ist die n Gleichungssysteme (fr je eine Spalte der Einheitsvektoren)
101 // in einem gemeinsamen Eliminationsverfahren.
102 // Statt nur einem Vektor fr die rechte Seite, erweitert man die Matrix um alle n Einheitsvektoren
103 // a11 a12 a13 | 1 0 0
104 // a21 a22 a23 | 0 1 0
105 // a31 a32 a33 | 0 0 1
106 // und eliminiert in der Matrix a alle Elemente ausserhalb der Diagonalen
107 // zuztlich sorgt man durch eine Division fr lauter 1 in der Diagonalen
108 // Das schaut dann so aus:
109 // x1 x2 x2
110 // 1 0 0 | b11 b12 b13
111 // 0 1 0 | b21 b22 b23
112 // 0 0 1 | b31 b32 b33
113 // Dadurch flt die Berechnung der Lsungen aus der Dreiecksmatrix (xn, xn-1, ... x1)
114 // weg, weil die Lsungen sofort abgelesen werden knnen.
115 // x1 = b11 bzw. b12 oder b13
116 // x2 = b21 bzw. b22 oder b23
117 // x3 = b31 bzw. b23 oder b33
118 // Das bedeutet, die Einheitsmatrix ist bei der Elimination in die Matrix B bergegangen,
119 // und das ist die Inverse Matrix.
120
121
122 int Inverse (double a[][NMAX], double ainvs[][NMAX], int n)
123 {
124     int i, j; // Zeile, Spalte
125     int s; // Eliminationsschritt
126     int pzeile; // Pivotzeile
127     int fehler = 0; // Fehlerflag

```

```

128 double f; // Multiplikationsfaktor
129 const double Epsilon = 0.001; // Genauigkeit
130 double Maximum; // Zeilenpivotisierung
131 extern FILE *fout;
132 int pivot = 1;
133
134 // ergnze die Matrix a um eine Einheitsmatrix (rechts anhggen)
135 for (i = 0; i < n; i++) {
136     for (j = 0; j < n; j++)
137     {
138         a[i][n+j] = 0.0;
139         if (i == j)
140             a[i][n+j] = 1.0;
141     }
142 }
143 #if DEBUG
144     MatOut (stdout, a, n, 2*n);
145 #endif
146
147 // die einzelnen Eliminationsschritte
148 s = 0;
149 do {
150     // Pivotisierung vermeidet unntigen Abbruch bei einer Null in der Diagonalen und
151     // erhht die Rechengenauigkeit
152     Maximum = fabs(a[s][s]);
153     if (pivot)
154     {
155         pzeile = s ;
156         for (i = s+1; i < n; i++)
157             if (fabs(a[i][s]) > Maximum) {
158                 Maximum = fabs(a[i][s]) ;
159                 pzeile = i;
160             }
161     }
162     fehler = (Maximum < Epsilon);
163
164     if (fehler) break; // nicht lsbar
165
166     if (pivot)
167     {
168         if (pzeile != s) // falls erforderlich, Zeilen tauschen
169         { double h;
170             for (j = s ; j < 2*n; j++) {
171                 h = a[s][j];
172                 a[s][j] = a[pzeile][j];
173                 a[pzeile][j]= h;
174             }
175         }
176     }
177
178     // Eliminationszeile durch Pivot-Koeffizienten  $f = a[s][s]$  dividieren
179     f = a[s][s];
180     for (j = s; j < 2*n; j++)
181         a[s][j] = a[s][j] / f;

```

```

182
183 // Elimination --> Nullen in Spalte s oberhalb und unterhalb der Diagonalen
184 // durch Addition der mit f multiplizierten Zeile s zur jeweiligen Zeile i
185 for (i = 0; i < n; i++) {
186     if (i != s)
187     {
188         f = -a[i][s]; // Multiplikationsfaktor
189         for (j = s; j < 2*n; j++) // die einzelnen Spalten
190             a[i][j] += f*a[s][j]; // Addition der Zeilen i, s
191     }
192 }
193 #if DEBUG
194 // fprintf(stdout, "Nach %i-tem Eliminationschritt:\n", s+1);
195     MatOut (stdout, a, n, 2*n);
196 #endif
197     s++;
198 } while ( s < n );
199
200 if (fehler)
201 {
202     fprintf (fout, "Inverse: Matrix ist singulär\n");
203     return 0;
204 }
205 // Die angehängte Einheitsmatrix Matrix hat sich jetzt in die inverse Matrix umgewandelt
206 // Umkopieren auf die Zielmatrix
207 for (i = 0; i < n; i++) {
208     for (j = 0; j < n; j++) {
209         ainv[i][j] = a[i][n+j];
210     }
211 }
212 return 1;
213 }
214
215 // Produkt
216 // Multipliziert die Matrix a mit der Matrix b , Resultat in Matrix c
217 // n ... Anzahl der Zeilen und Spalten
218 void Produkt (double a[][NMAX], double b[][NMAX], double c[][NMAX], int n)
219 {
220     int i, j, k;
221     for (i = 0; i < n; i++)
222     {
223         for (j = 0; j < n; j++)
224         {
225             c[i][j] = 0.0;
226             for (k = 0; k < n; k++)
227                 c[i][j] += a[i][k]*b[k][j];
228         }
229     }
230 } // end Produkt
231
232
233
234 // Einfache Eingabe einer Matrix, allgemein gehalten
235 // fr Gleichungssysteme ist m = n + 1

```

```

236 void MatIn (FILE *f, double a[][NMAX], int *n, int *m)
237 {
238     int bFile = 1;
239     int i, j;
240
241     if (f == stdin)
242         bFile = 0;
243
244     if (!bFile) {
245         printf("Matrix_Eingabe_\n\n");
246         printf("Anzahl_der_Zeilen_(<%2i)_n_=_", NMAX);
247     }
248     fscanf (f, "%d", n);
249     if (!bFile) printf("Anzahl_der_Spalten_(<%2i)_m_=_", NMAX);
250     fscanf (f, "%d", m);
251     if (!bFile) printf("\n");
252
253     for (i = 0; i < *n; i++) {
254         if (!bFile) printf("Zeile_%2d:_", i+1);
255         for (j = 0; j < *m; j++)
256             fscanf(f, "%lf", &a[i][j]);
257     }
258
259     if (!bFile) fgetc(f);
260     //fprintf(f, "\n");
261
262 }
263
264 void MatOut (FILE *f, double a[][NMAX], int n, int m)
265 {
266     int bFile = 1;
267     int i, j;
268
269     if (f == stdout)
270         bFile = 0;
271
272     fprintf(f, "\n");
273     for (i = 0; i < n; i++) {
274         for (j = 0; j < m; j++)
275             fprintf (f, "%7.7f_", a[i][j]);
276         fprintf(f, "\n");
277     }
278     fprintf(f, "
-----
n");
279     if (!bFile) {
280         // printf("Weiter mit Eingabetaste");
281         // getchar();
282     }
283 }

```

4 Räumliches Gelenkviereck

4.1 Übung

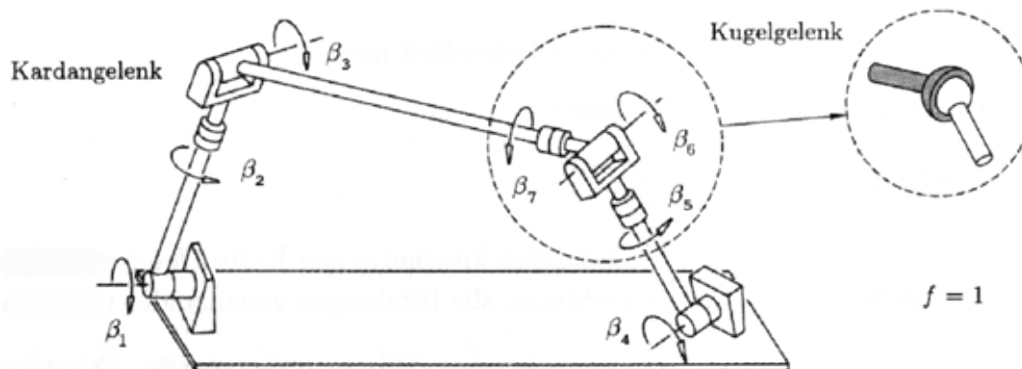


Abbildung 4.1: räumliches Gelenkviereck

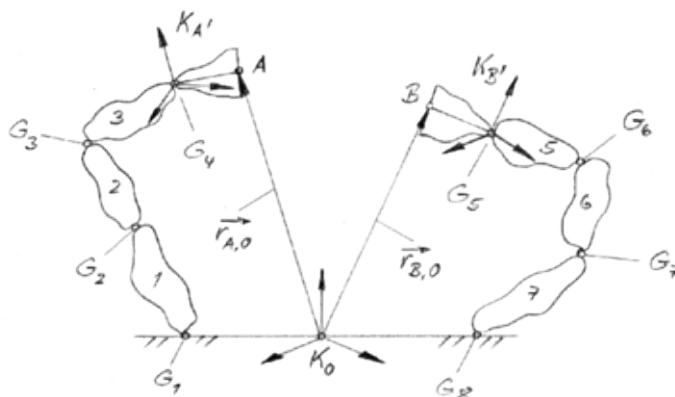


Abbildung 4.2: Körperschnitt bei einer räumlichen kinematischen Schleife

Ziel dieser Übung ist es, eine räumliche kinematische Schleife, wie in Abbildung 4.1 dargestellt, zu animieren. Wurde in Aufgabe 3 nur um die z-Achse gedreht, werden hier weitere Achsen verwendet. Durch einen Schnitt am Gelenkpaar in den Ursprüngen von K_3 und K_6 werden für diskrete Gelenkwinkel θ_1 die zugehörigen Gelenkwinkel θ_4 iteriert.

Ein weiterer Schnitt am Kugelgelenk sorgt für die Iteration der Gelenkwinkel θ_2 und θ_3 . Mittels eines zusätzlichen Schnitts am Drehgelenk G_7 werden anschließend die noch fehlenden Gelenkwinkel θ_5 und θ_6 iteriert. Zur besseren Darstellung ist jeder Körper und jedes Drehgelenk plastisch. Kreuzgelenke sind durch zusätzlich angebrachte Körper in ihrer Gestalt erkennbar.

4.2 Lösungsansatz

Die Techniken wurden bereits in den vorherigen Kapiteln erläutert, daher werden hier nur die Funktionen aufgezählt.

Die Funktion `void zeichneKette()` bestimmt zunächst die vorgegebenen Werte und löst die Berechnungen aus. Anschließend gibt sie die komplette Szene wieder. Die Teilketten sind wieder in eigenen `glPushMatrix-/glPopMatrix`-Konstrukten eingeschlossen, weil sie voneinander unabhängig sind. Ihre Berechnung geht aber gemeinsam von K_0 aus (Abb. 4.2). Die Berechnungen sind mit entsprechenden Kommentaren in der Datei `newton.cpp`, aufgeführt.

4.2.1 Starrkörper

Um die Animation besser erkennbar zu machen, werden keine Linien, sondern Zylinder als Starrkörper dargestellt. Da die Zylinder aus Linien bestehen, die mithilfe von Rotation um eine feste Koordinate im Raum angeordnet und die Zylinder nach Bedarf unterschiedlich lang und in unterschiedliche Richtungen ausgerichtet sind, ist die Funktion `void zylinder(float _laenge, float _radius, float _iterationen, char _seite)` generisch gehalten.

4.2.2 Aufbau

Das Modell wird zunächst in seiner Ausgangsposition gezeichnet. In jedem Schritt (Änderung von θ_1 um den Wert 0,01) werden die Winkel θ_{2-6} durch Nullstelleniteration ermittelt. Dann wird die linke Teilkette gezeichnet. Auch hier wird wieder ausgenutzt, daß innerhalb eines `glPushMatrix-/glPopMatrix`-Konstruktes sämtliche Translationen aufeinander aufbauen und somit die hierarchische Anordnung der Körper und Gelenke eingehalten wird. Ist das erledigt, wird, wieder vom Inertialsystem ausgehend, die rechte Teilkette gezeichnet und trifft sich mit dem Ende der linken Teilkette.

4.2.3 Berechnung

Es werden wieder Gleichungen des transzendenten Gleichungssystems verwendet. Diese heißen `double f1(double wert)`, `double f2(double t1, double t2, double t3, double t4)`, `double f3 (double t1, double t2, double t3, double t4)`, `double f4(double t1, double t4, double t5, double t6)` und `double f5(double t1, double t4, double t5, double t6)`. Ferner gibt es `f1s(double wert)`, die numerische Ableitung der Funktion `f1(double wert)`.

Die Jacobimatrix für die zweite Iteration aus den Funktionen `f3` und `f4` wird in der Funktion `void jacobiMatrixIteration_2(double t1, double t2, double t3, double t4, double matrix[2][4])` erzeugt, und die Jacobimatrix für die dritte Iteration wird in `void jacobiMatrixIteration_3(double t1, double t2, double t3, double t4, double matrix[2][4])` erzeugt.

Mit `berechneWinkel()` werden die Iterationen durchgeführt, um θ_4 , θ_2 und θ_3 , θ_5 und θ_6 in Abhängigkeit mit θ_1 zu bestimmen.

4.2.4 Bedienung

Die Animation wird mit d gestartet und angehalten oder mit den Pfeiltasten links und rechts manuell gesteuert. Beim Drücken der Leertaste ändert sich die jeweilige Drehrichtung von G_1 . Mit q kann das Programm beendet werden. Die Gitterbox mit der darin befindlichen kinematischen Schleife kann bei gedrückter Maustaste um seine x- und z-Achse per Mausbewegung gedreht werden.

4.3 Darstellung



Abbildung 4.3: linke Teilkette



Abbildung 4.4: rechte Teilkette



Abbildung 4.5: kinematische Schleife

4.4 Quellcode

```
1 #include <cstdlib>
2 #include <iostream>
3 #include <GL/gl.h>
4 #include <GL/glut.h>
5 #include <cmath>
6 #include "newton.h"
7 #include "koerper.h"
8
9 using namespace std;
10
11 bool demo = false;
12 bool vorwaerts = false;
13 float thickness = 5.0;
14
15 // Winkel, in denen die Gelenkpaare zueinander stehen
16 double beta1 = -100.0 * M_PI / 180.0; double theta1 = 30.0 * M_PI / 180.0;
17 double theta2 = 2.0;
18 double theta3 = 1.0;
19 double beta4 = -80.0 * M_PI / 180.0; double gamma4 = 50.0 * M_PI / 180.0; double theta4 = 3.0;
20 double theta5 = 1.0;
21 double theta6 = -1.5;
22 double theta7 = 0.0;
23
24 double xOld, yOld;
25
26 GLfloat mouseAngleX = 90.0;
27 GLfloat mouseAngleY = 180.0;
28 GLfloat mouseAngleZ = 0.0;
29
30 void init();
31 void display();
32 void keyboard(unsigned char key, int x, int y);
33 void specialKey(int key, int x, int y);
34 void motion(int x, int y);
35 void passiveMotion(int x, int y);
36 void idle();
37 void zeichneKette();
38
39 void matrizenProdukt(double ergebnis[3][3], double matrix1[3][3], double matrix2[3][3])
40 {
41     for (int zeile = 0; zeile < 3; zeile++)
42     {
43         for (int spalte = 0; spalte < 3; spalte++)
44         {
45             ergebnis[zeile][spalte] = matrix1[zeile][0] * matrix2[0][spalte] +
46                                     matrix1[zeile][1] * matrix2[1][spalte] +
47                                     matrix1[zeile][2] * matrix2[2][spalte];
48         }
49     }
50 }
51
52 void matrixVektorProdukt(double ergebnis[3], double matrix[3][3], double vektor[3])
```

```

53 {
54     for (int zeile = 0; zeile < 3; zeile++)
55     {
56         ergebnis[zeile] = matrix[zeile][0] + vektor[zeile] +
57                             matrix[zeile][1] + vektor[zeile] +
58                             matrix[zeile][2] + vektor[zeile];
59     }
60 }
61
62 void vektorAddition(double ergebnis[3], double vektor1[3], double vektor2[3])
63 {
64     for (int zeile = 0; zeile < 3; zeile++)
65     {
66         ergebnis [zeile] = vektor1[zeile] + vektor2[zeile];
67     }
68 }
69
70 int main(int argc, char *argv[])
71 {
72     glutInit(&argc, argv);
73     glutInitDisplayMode(GLUT_RGB | GLUT_DOUBLE | GLUT_DEPTH);
74     glutInitWindowSize(700, 700);
75     glutCreateWindow(" raeumliche_kinematische_Kette");
76     glutDisplayFunc(display);
77     glutKeyboardFunc(keyboard);
78     glutSpecialFunc(specialKey);
79     glutMotionFunc(motion);
80     glutPassiveMotionFunc(passiveMotion);
81     init();
82     glutIdleFunc(idle);
83     glutMainLoop();
84     return EXIT_SUCCESS;
85 }
86
87 void init()
88 {
89     // Materialeigenschaften
90     GLfloat _ambient[] = {0.25, 0.25, 0.25};
91     GLfloat _diffuse[] = {0.2, 0.2, 0.24};
92     GLfloat _specular[] = {0.4, 0.4, 0.4};
93     GLfloat _emission[] = {0.2, 0.2, 0.2};
94     GLfloat _shininess[] = {30.0};
95
96     glEnable(GL_DEPTH_TEST);
97     glEnable(GL_COLOR_MATERIAL);
98     glClearDepth(1.0);
99     glClearColor(1.0, 1.0, 1.0, 1.0);
100
101     glMaterialfv (GL_FRONT, GL_AMBIENT, _ambient);
102     glMaterialfv (GL_FRONT, GL_DIFFUSE, _diffuse);
103     glMaterialfv (GL_FRONT, GL_SPECULAR, _specular);
104     glMaterialfv (GL_FRONT, GL_SHININESS, _shininess);
105     glMaterialfv (GL_FRONT, GL_EMISSION, _emission);
106

```

```

107     glEnable(GL_LIGHTING);
108     glEnable(GL_LIGHT0);
109
110     // Rotes Licht, da sonst berblendet
111     const GLfloat lightPosition[] = {0.0, 100.0, -1.0};
112     const GLfloat light_ambient[] = {0.1, 0.0, 0.0};
113     const GLfloat light_diffuse[] = {0.1, 0.0, 0.0};
114     const GLfloat light_specular[] = {0.1, 0.0, 0.0};
115
116     glLightfv(GL_LIGHT0, GL_POSITION, lightPosition);
117     glLightfv(GL_LIGHT0, GL_AMBIENT, light_ambient);
118     glLightfv(GL_LIGHT0, GL_DIFFUSE, light_diffuse);
119     glLightfv(GL_LIGHT0, GL_SPECULAR, light_specular);
120 }
121
122 void display()
123 {
124     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
125
126     // Projektions-Stack
127     glMatrixMode(GL_PROJECTION);
128     glLoadIdentity();
129     glOrtho(-200.0, 200.0, -200.0, 200.0, -300.0, 300.0);
130
131     glMatrixMode(GL_MODELVIEW);
132     glLoadIdentity();
133
134     // mit Maus steuerbarer Blickwinkel
135     glPushMatrix();
136     glRotatef(mouseAngleX, 1.0, 0.0, 0.0);
137     glRotatef(mouseAngleY, 0.0, 1.0, 0.0);
138     glRotatef(mouseAngleZ, 0.0, 0.0, 1.0);
139     zeichneKette();
140     glPopMatrix();
141     glutSwapBuffers();
142 }
143
144 void zeichneKette()
145 {
146     double matrix_tmp[3][3];
147     double vektor_tmp[3];
148
149     // Ansichts-Eigenschaften des Kugelgelenks
150     GLUquadric *qobj = gluNewQuadric();
151     gluQuadricDrawStyle(qobj, GLU_FILL);
152     gluQuadricNormals(qobj, GLU_FLAT);
153     gluQuadricOrientation(qobj, GLU_OUTSIDE);
154
155     // Einheitsmatrix fuer T01alpha1, T01gamma1, T01alpha4, T01gamma4, T1a2, T2a3, T4a5, T5a6,
156     // T6a7
157     double einheitsmatrix[3][3] = {{1.0, 0.0, 0.0},
158                                     {0.0, 1.0, 0.0},
159                                     {0.0, 0.0, 1.0}};

```

```

160 // Geometriedaten G1 bis G6'
161 double x01 = 100.0; // 0x1-Komponente von G1
162 double y01 = -20.0; // 0y1-Komponente von G1
163 double z01 = 30.0; // 0z1-Komponente von G1
164 double x1a2 = 40.0; // 1'z2-Komponente von G2
165 double x2a3 = 20.0; // 2'x3-Komponente von G3
166 double x3a6a = 200.0; // 3'x6-Komponente von G6
167 double alpha1 = 0.0; // Drehung K0 um x0-Achse, 1. Drehung
168 double gamma1 = 0.0; // Drehung K0 um gedrehte z0-Achse, 3. Drehung
169
170 double T01beta1[3][3] = {{cos(beta1), 0.0, sin(beta1)},
171                          {0.0, 1.0, 0.0},
172                          {-sin(beta1), 0.0, cos(beta1)}};
173 // T01 = T01alpha1 * T01beta1 * t01gamma1
174 double T01[3][3]; // Drehtransformation der Koordinaten K1 in K0
175 matrizenProdukt(matrix_tmp, einheitsmatrix, T01beta1);
176 matrizenProdukt(T01, matrix_tmp, einheitsmatrix);
177
178 // Geometriedaten G4 bis G6
179 double x04 = -140.0; // 0y4-Komponente von G4
180 double y04 = 10.0; // 0y4-Komponente von G4
181 double z04 = 30.0; // 0z4-Komponente von G4
182 double x4a5 = 140.0; // 4'x5-Komponente von G5
183 double x5a6 = 20.0; // 5'x6-Komponente von G6
184 double x6a7 = 20.0; // 6'x7-Komponente von G7
185 double x7a3a = 180.0; // 7'x3-Komponente von G3
186 double x6a3a = 200.0; // 6'x3-Komponente von G3
187 double alpha4 = 0.0; // Drehung um x0-Achse, 1. Drehung
188 double T01beta4[3][3] = {{cos(beta4), 0.0, sin(beta4)},
189                          {0.0, 1.0, 0.0},
190                          {-sin(beta4), 0.0, cos(beta4)}};
191 double T01gamma4[3][3] = {{cos(gamma4), -sin(gamma4), 0.0},
192                          {sin(gamma4), cos(gamma4), 0.0},
193                          {0.0, 0.0, 1.0}};
194 // T04 = T01alpha4 * t01gamma4 * t01beta4
195 double T04[3][3]; // Drehtransformation der Koordinaten K4 in K0
196 matrizenProdukt(matrix_tmp, einheitsmatrix, T01gamma4);
197 matrizenProdukt(T04, matrix_tmp, T01beta4);
198
199 // Theta 2-6 errechnen
200
201 berechneWinkel();
202
203 // Vorwärtskinematik linke Teilkette bei Schnitt am Gelenkpaar
204
205 double r010[3] = {x01, y01, z01}; // Vektor vom Ursprung K0 nach K1 in K0
206 double r1a31a[3] = {x1a2 + x2a3, 0.0, 0.0}; // Vektor vom Ursprung K1' nach K3 in K1'
207 double T11a[3][3] = {{cos(theta1), -sin(theta1), 0.0},
208                     {sin(theta1), cos(theta1), 0.0},
209                     {0.0, 0.0, 1.0}};
210 // T01a = T01 * T11a
211 double T01a[3][3];
212 matrizenProdukt(T01a, T01, T11a);
213 // r030Pa = r010 + T01a * r1a31a

```

```

214 double r030Pa[3]; // Vektor vom Ursprung K0 nach K3 in K0 bei Schnitt am Gelenkpaar
215 matrixVektorProdukt(vektor_tmp, T01a, r1a31a);
216 vektorAddition(r030Pa, r010, vektor_tmp);
217
218 // Ergaenzg. Vorwaertskinetik linke Teilkette bei Schnitt am Kugelgelenk
219
220 double r1a21a[3] = {x1a2, 0.0, 0.0}; // Vektor vom Ursprung K1' nach K2 in K1'
221 double r2a32a[3] = {x2a3, 0.0, 0.0}; // Vektor vom Ursprung K2' nach K3 in K2'
222 double r3a6a3a[3] = {x3a6a, 0.0, 0.0}; // Vektor vom Ursprung K3' nach K6' in K3'
223 double T22a[3][3] = {{1.0, 0.0, 0.0},
224                     {0.0, cos(theta2), -sin(theta2)},
225                     {0.0, sin(theta2), cos(theta2)}};
226 // T02a = T01a * T1a2 * T22a
227 double T02a[3][3];
228 matrizenProdukt(matrix_tmp, T01a, einheitsmatrix);
229 matrizenProdukt(T02a, matrix_tmp, T22a);
230
231 double T33a[3][3] = {{cos(theta3), -sin(theta3), 0.0},
232                     {sin(theta3), cos(theta3), 0.0},
233                     {0.0, 0.0, 1.0}};
234 // T03a = T02a * T2a3 * T33a
235 double T03a[3][3];
236 matrizenProdukt(matrix_tmp, T02a, einheitsmatrix);
237 matrizenProdukt(T03a, matrix_tmp, T33a);
238 // r020 = r010 + T01a * r1a21a
239 double r020[3];
240 matrixVektorProdukt(vektor_tmp, T01a, r1a21a);
241 vektorAddition(r020, r010, vektor_tmp);
242 // r030Ku = r020 + T02a * r2a32a
243 double r030Ku[3];
244 matrixVektorProdukt(vektor_tmp, T02a, r2a32a);
245 vektorAddition(r030Ku, r020, vektor_tmp);
246 // r06a0 = r030Ku + T03a * r3a6a3a
247 double r06a0[3];
248 matrixVektorProdukt(vektor_tmp, T03a, r3a6a3a);
249 vektorAddition(r06a0, r030Ku, vektor_tmp);
250
251 // r030Ku = r010 + T01a * r1a31a
252 matrixVektorProdukt(vektor_tmp, T01a, r1a31a);
253 vektorAddition(r030Ku, r010, vektor_tmp);
254
255 // Vorwaertskinetik rechte Teilkette bei Schnitt am Gelenkpaar bzw. Kugelgelenk
256
257 double r040[3] = {x04, y04, z04}; // Vektor vom Ursprung K0 nach K4 in K0
258 double r4a64a[3] = {x4a5 + x5a6, 0.0, 0.0}; // Vektor vom Ursprung K4' nach K6 in K4'
259 double T44a[3][3] = {{cos(theta4), -sin(theta4), 0.0},
260                     {sin(theta4), cos(theta4), 0.0},
261                     {0.0, 0.0, 1.0}};
262 // T04a = T04 * T44a
263 double T04a[3][3];
264 matrizenProdukt(T04a, T04, T44a);
265 // r060 = r040 + T04a * r4a64a
266 double r060[3]; // Vektor vom Ursprung K0 nach K6 in K0
267 matrixVektorProdukt(vektor_tmp, T04a, r4a64a);

```

```

268 vektorAddition(r060, r040, vektor_tmp);
269
270 // Ergaenzung Vorwaertskinematik rechte Teilkette bei Schnitt am Drehgelenk G7
271
272 double r4a54a[3] = {x4a5, 0.0, 0.0}; // Vektor vom Ursprungs K4' nach K5 in K4'
273 double r5a65a[3] = {x5a6, 0.0, 0.0}; // Vektor vom Ursprung K5' nach K6 in K5'
274 double r6a76a[3] = {x6a7, 0.0, 0.0}; // Vektor vom Ursprung K6' nach K7 in K6'
275 double r7a3a7a[3] = {x7a3a, 0.0, 0.0}; // Vektor vom Ursprung K7' nach K3' in K7'
276
277 double T55a[3][3] = {{1.0, 0.0, 0.0},
278                    {0.0, cos(theta5), -sin(theta5)},
279                    {0.0, sin(theta5), cos(theta5)}};
280 // T05a = T04a * T4a5 * T55a
281 double T05a[3][3];
282 matrizenProdukt(matrix_tmp, T04a, einheitsmatrix);
283 matrizenProdukt(T05a, matrix_tmp, T55a);
284
285 double T66a[3][3] = {{cos(theta6), -sin(theta6), 0.0},
286                    {sin(theta6), cos(theta6), 0.0},
287                    {0.0, 0.0, 1.0}};
288 // T06a = T05a * T5a6 * T66a
289 double T06a[3][3];
290 matrizenProdukt(matrix_tmp, T05a, einheitsmatrix);
291 matrizenProdukt(T06a, matrix_tmp, T66a);
292
293 double T77a[3][3] = {{1.0, 0.0, 0.0},
294                    {0.0, cos(theta7), -sin(theta7)},
295                    {0.0, sin(theta7), cos(theta7)}};
296 // T07a = T06a * T6a7 * T77a
297 double T07a[3][3];
298 matrizenProdukt(matrix_tmp, T06a, einheitsmatrix);
299 matrizenProdukt(T07a, matrix_tmp, T77a);
300 // r050 = r040 + T04a * r4a64a
301 double r050[3];
302 matrixVektorProdukt(vektor_tmp, T04a, r4a54a);
303 vektorAddition(r050, r040, vektor_tmp);
304 // r060Kr = r050 + T05a * r5a65a
305 double r060Kr[3];
306 matrixVektorProdukt(vektor_tmp, T05a, r5a65a);
307 vektorAddition(r060Kr, r050, vektor_tmp);
308 // r070 = v070 * T07a * r6a76a
309 double r070[3];
310 matrixVektorProdukt(vektor_tmp, T06a, r6a76a);
311 vektorAddition(r070, r060Kr, vektor_tmp);
312 // r03a0 = r070 + T07a * r7a3a7a
313 double r03a0[3];
314 matrixVektorProdukt(vektor_tmp, T07a, r7a3a7a);
315 vektorAddition(r03a0, r070, vektor_tmp);
316
317 // Darstellung der geometrischen Lagen
318
319 double r01u[3] = {x01, y01, 0.0}; // Vektor vom Ursprung K0 zum Lotpunkt K1 in K0
320 double r04u[3] = {x04, y04, 0.0}; // Vektor vom Ursprung K0 zum Lotpunkt K4 in K0
321

```

```

322 double gz_tmp1[3] = {0.0, 0.0, 10.0}; // temporaerer Vektor
323 double gz_tmp2[3] = {0.0, 0.0, -10.0}; // temporaerer Vektor
324 // gz11 = r010 + T01a * gz_tmp1
325 double gz11[3]; // Drehachse von Gelenk 1 = z1-Achse
326 matrixVektorProdukt(vektor_tmp, T01a, gz_tmp1);
327 vektorAddition(gz11, r010, vektor_tmp);
328 // gz12 = r010 + T01a * gz_tmp2
329 double gz12[3]; // Drehachse von Gelenk 1 = z1-Achse
330 matrixVektorProdukt(vektor_tmp, T01a, gz_tmp2);
331 vektorAddition(gz12, r010, vektor_tmp);
332 // gz41 = r040 + T04a * gz_tmp1
333 double gz41[3]; // Drehachse von Gelenk 4
334 matrixVektorProdukt(vektor_tmp, T04a, gz_tmp1);
335 vektorAddition(gz41, r040, vektor_tmp);
336 // gz42 = r040 + T04a * gz_tmp2
337 double gz42[3]; // Drehachse von Gelenk 4
338 matrixVektorProdukt(vektor_tmp, T04a, gz_tmp2);
339 vektorAddition(gz42, r040, vektor_tmp);
340 // achseKu11 = r020 + T01a * gz_tmp1
341 double achseKu11[3]; // z2-Drehachse von Gelenk 2
342 matrixVektorProdukt(vektor_tmp, T01a, gz_tmp1);
343 vektorAddition(achseKu11, r020, vektor_tmp);
344 // achseKu12 = r020 + T01a * gz_tmp2
345 double achseKu12[3]; // z2-Drehachse von Gelenk 2
346 matrixVektorProdukt(vektor_tmp, T01a, gz_tmp2);
347 vektorAddition(achseKu12, r020, vektor_tmp);
348 // achseKu21 = r020 + T02a * gz_tmp1
349 double achseKu21[3]; // z2-Drehachse von Gelenk 2
350 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp1);
351 vektorAddition(achseKu21, r020, vektor_tmp);
352 // achseKu22 = r020 + T02a * gz_tmp1
353 double achseKu22[3]; // z2-Drehachse von Gelenk 2
354 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp2);
355 vektorAddition(achseKu22, r020, vektor_tmp);
356 // achseKu31 = r030Ku + T02a * gz_tmp1
357 double achseKu31[3]; // z2'-Drehachse von Gelenk 2
358 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp1);
359 vektorAddition(achseKu31, r030Ku, vektor_tmp);
360 // achseKu32 = r030Ku + T02a * gz_tmp2
361 double achseKu32[3]; // z2'-Drehachse von Gelenk 2
362 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp2);
363 vektorAddition(achseKu32, r030Ku, vektor_tmp);
364 // achseKr11 = r070 + T02a * gz_tmp1
365 double achseKr11[3]; // z2'-Drehachse von Gelenk 2
366 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp1);
367 vektorAddition(achseKr11, r070, vektor_tmp);
368 // achseKr12 = r070 + T02a * gz_tmp2
369 double achseKr12[3]; // z2'-Drehachse von Gelenk 2
370 matrixVektorProdukt(vektor_tmp, T02a, gz_tmp2);
371 vektorAddition(achseKr12, r070, vektor_tmp);
372 // achseKr21 = r070 + T06a * gz_tmp1
373 double achseKr21[3]; // z2'-Drehachse von Gelenk 2
374 matrixVektorProdukt(vektor_tmp, T06a, gz_tmp1);
375 vektorAddition(achseKr21, r060Kr, vektor_tmp);

```

```

376 // achseKr22 = r070 + T06a * gz_tmp2
377 double achseKr22[3]; // z2'-Drehachse von Gelenk 2
378 matrixVektorProdukt(vektor_tmp, T06a, gz_tmp2);
379 vektorAddition(achseKr22, r060Kr, vektor_tmp);
380 // achseKr31 = r070 + T04 * gz_tmp1
381 double achseKr31[3]; // z2'-Drehachse von Gelenk 2
382 matrixVektorProdukt(vektor_tmp, T04, gz_tmp1);
383 vektorAddition(achseKr31, r050, vektor_tmp);
384 // achseKr32 = r070 + T04 * gz_tmp2
385 double achseKr32[3]; // z2'-Drehachse von Gelenk 2
386 matrixVektorProdukt(vektor_tmp, T04, gz_tmp2);
387 vektorAddition(achseKr31, r050, vektor_tmp);
388
389 glColor3f(1.0, 1.0, 1.0);
390
391 // Platte
392 glPushMatrix();
393   glTranslatef(-20.0, 10.0, -75.0);
394   glBegin(GL_QUADS);
395     glVertex3f(-150.0, 150.0, 0.0);
396     glVertex3f(150.0, 150.0, 0.0);
397     glVertex3f(150.0, -150.0, 0.0);
398     glVertex3f(-150.0, -150.0, 0.0);
399   glEnd();
400 glPopMatrix();
401
402 // Kaefig
403 glPushMatrix();
404   glTranslatef(-20.0, 10.0, 0.0);
405   glutWireCube(300);
406 glPopMatrix();
407
408
409 glPushMatrix();
410   glTranslatef(0.0, 0.0, -75.0);
411
412   // linke Teilkette
413   glPushMatrix();
414     glColor3f(0.2, 0.2, 0.2);
415     glTranslatef(r01u[0], r01u[1], r01u[2]);
416     zylinder(30.0, thickness + 10, 4, 'z');
417 // zylinder(10.0, thickness + 10, 4, 'z');
418 // achse('x');
419     glTranslatef(0.0, 0.0, 30.0);
420     glColor3f(1.0, 0.0, 0.0);
421
422     // G1
423     glRotatef(beta1 * 180/M_PI, 0.0, 1.0, 0.0);
424     glRotatef(theta1 * 180/M_PI, 0.0, 0.0, 1.0);
425     glTranslatef(0.0, 0.0, -10.0);
426     zylinder(20.0, thickness, 50, 'z');
427     glTranslatef(0.0, 0.0, 10.0);
428
429     // K1

```

```

430     zylinder(40.0, thickness, 50, 'x');
431     glTranslatef(40.0, 0.0, 0.0);
432
433     // G2
434     glTranslatef(0.0, 0.0, -10.0);
435     zylinder(20.0, thickness, 50, 'z');
436     glTranslatef(0.0, 0.0, 10.0);
437
438     glRotatef(theta2 * 180/M_PI, 1.0, 0.0, 0.0);
439     glTranslatef(0.0, 0.0, -10.0);
440     zylinder(20.0, thickness, 50, 'z');
441     glTranslatef(0.0, 0.0, 10.0);
442
443     // Drehaufsatz K1
444     zylinder(20.0, thickness, 50, 'x');
445     glTranslatef(20.0, 0.0, 0.0);
446
447     // G3
448     gluSphere(qobj, thickness * 1.5, 40, 40);
449     glTranslatef(0.0, 0.0, -10.0);
450     zylinder(20.0, thickness, 50, 'z');
451     glTranslatef(0.0, 0.0, 10.0);
452     glRotatef(theta3 * 180/M_PI, 0.0, 0.0, 1.0);
453
454     // K2
455     zylinder(180.0, thickness, 50, 'x');
456     glTranslatef(180.0, 0.0, 0.0);
457
458     // G7
459     glTranslatef(0.0, 0.0, -10.0);
460     zylinder(20.0, thickness, 50, 'z');
461     glPopMatrix();
462
463     // rechte Teilkette
464     glPushMatrix();
465     glColor3f(0.2, 0.2, 0.2);
466     glTranslatef(r04u[0], r04u[1], r04u[2]);
467     zylinder(30.0, thickness + 10, 4, 'z');
468     glTranslatef(0.0, 0.0, 30.0);
469
470     // G4
471     glColor3f(1.0, 0.0, 0.0);
472     glRotatef(gamma4 * 180/M_PI, 0.0, 0.0, 1.0);
473     glRotatef(beta4 * 180/M_PI, 0.0, 1.0, 0.0);
474     glRotatef(theta4 * 180/M_PI, 0.0, 0.0, 1.0);
475     achse('z');
476     glTranslatef(0.0, 0.0, -10.0);
477     zylinder(20.0, thickness, 50, 'z');
478     glTranslatef(0.0, 0.0, 10.0);
479
480     // K3
481     zylinder(140.0, thickness, 50, 'x');
482     glTranslatef(140.0, 0.0, 0.0);
483

```

```

484 // G5
485 glTranslatef(0.0, 0.0, -10.0);
486 cylinder(20.0, thickness, 50, 'z');
487 glTranslatef(0.0, 0.0, 10.0);
488 glRotatef(theta5 * 180/M_PI, 1.0, 0.0, 0.0);
489
490 // Drehaufsatz K3
491 cylinder(20.0, thickness, 50, 'x');
492 glTranslatef(20.0, 0.0, 0.0);
493
494 // G6
495 glRotatef(theta6 * 180/M_PI, 0.0, 0.0, 1.0);
496 gluSphere(qobj, thickness * 1.5, 40, 40);
497
498 //Drehaufsatz K2
499 glRotatef(theta7 * 180/M_PI, 1.0, 0.0, 0.0);
500 cylinder(20.0, thickness, 50, 'x');
501 glPopMatrix();
502 glPopMatrix();
503 }
504
505 void keyboard(unsigned char key, int x, int y)
506 {
507     if (key == 'q') exit(0);
508     if (key == 'd') demo = !demo;
509     if (key == '\n') vorwaerts = !vorwaerts;
510 }
511
512 void specialKey(int key, int x, int y)
513 {
514     bool keyPressed = false;
515     switch(key)
516     {
517         case GLUT_KEY_LEFT: theta1 += 0.1; keyPressed = true; break;
518         case GLUT_KEY_RIGHT: theta1 -= 0.1; keyPressed = true; break;
519         case GLUT_KEY_UP: thickness += 0.1; keyPressed = true; break;
520         case GLUT_KEY_DOWN: thickness -= 0.1; keyPressed = true; break;
521     }
522     if (keyPressed) glutPostRedisplay();
523 }
524
525 void motion(int x, int y)
526 {
527     mouseAngleX = fmod(mouseAngleX + 0.3 * (y - yOld) + 1800.0, 360.0);
528     mouseAngleZ = fmod(mouseAngleZ + 0.3 * (x - xOld) + 1800.0, 360.0);
529     xOld = x;
530     yOld = y;
531     glutPostRedisplay();
532 }
533
534 void passiveMotion(int x, int y)
535 {
536     xOld = x;
537     yOld = y;

```

```

538 }
539
540 void idle()
541 {
542     if (demo)
543     {
544         if (vorwaerts) theta1+= 0.01;
545         else theta1-= 0.01;
546         glutPostRedisplay();
547     }
548 }

```

newton.cpp:

```

1  #include <cmath>
2
3  double grad2Rad(double wert);
4  double g1(double wert);
5  double g1s(double wert);
6  double g2(double t1, double t2, double t3, double t4);
7  double g3(double t1, double t2, double t3, double t4);
8  double g4(double t1, double t4, double t5, double t6);
9  double g5(double, double, double, double);
10 int inverseMatrix(double a[2][4], double ainv[2][4], int n);
11 double partielleAbleitung(double (*f)(double, double, double, double), double t1, double t2, double t3,
    double t4, int pnr);
12 void berechneWinkel();
13 void jacobiMatrixIteration_2(double t1, double t2, double t3, double t4, double matrix[2][4]);
14 void jacobiMatrixIteration_3(double t1, double t2, double t3, double t4, double matrix[2][4]);
15
16 extern double theta1, theta2, theta3, theta4, theta5, theta6, beta1, beta4, gamma4;
17
18 using namespace std;
19
20 double grad2Rad(double wert)
21 {
22     return wert * M_PI / 180;
23 }
24
25 double g1(double wert)
26 {
27     double beta1 = -100.0 * M_PI / 180;
28     double beta4 = -80.0 * M_PI / 180;
29     double gamma4 = 50.0 * M_PI / 180;
30     double ergebnis = sqrt(
31         pow((-160 * cos(wert) * sin(beta4) + 60 * cos(theta1) * sin(beta1)), 2)
32         + pow((-240 - 60 * cos(beta1) * cos(theta1) + 160 * (cos(gamma4) * cos(beta4) * cos(
    wert) - sin(gamma4) * sin(wert))), 2)
33         + pow((30 + 160 * (cos(beta4) * cos(wert) * sin(gamma4) + cos(gamma4) * sin(wert)) -
    60 * sin(theta1)), 2)) - 200;
34     return ergebnis;
35 }
36
37 double g2(double t1, double t2, double t3, double t4)

```

```

38 {
39     double ergebnis = 30 + 160 *(sin(gamma4) * cos(beta4) * cos(t4) + cos(gamma4) * sin(t4)) - 60 *
        sin(t1) - 200 * ( sin(t1) * cos(t3) + cos(t1) * cos(t2) * sin(t3));
40     return ergebnis;
41 }
42
43 double g3(double t1, double t2, double t3, double t4)
44 {
45     double ergebnis = - 160 * sin(beta4) * cos(t4) + 60 * sin(beta1) * cos(t1) - 200 * (- sin(beta1) *
        cos(t1) * cos(t3) + (cos(t2) * sin(beta1) * sin(t1) + cos(beta1) * sin(t2)) * sin(t3) );
46     return ergebnis;
47 }
48
49 double g4(double t1, double t4, double t5, double t6)
50 {
51     double ergebnis = 10 + 160 * (cos(beta4) * cos(t4) * sin(gamma4) + cos(gamma4)*sin(t4)) + 200 *
        (cos(t6) * ( cos(beta4) * cos(t4) * sin(gamma4) + cos(gamma4) * sin(t4)) + (cos(t5) * (cos(
        gamma4) * cos(t4) - cos(beta4) * sin(gamma4) * sin(t4)) + sin(beta4) * sin(gamma4) * sin(t5)
        ) * sin(t6)) - (-20 + 60 * sin(t1));
52     return ergebnis;
53 }
54
55 double g5(double t1, double t4, double t5, double t6)
56 {
57     double ergebnis = 30 - 160 * cos(t4) * sin(beta4) + 200 * (-cos(t4) * cos(t6) * sin(beta4) + (cos(
        t5) * sin(beta4) * sin(t4) + cos(beta4) * sin(t5)) * sin(t6)) - (30 - 60 * cos(t1)* sin(beta1));
58     return ergebnis;
59 }
60
61 // numerische Ableitung von Gleichung g1
62 double g1s(double wert)
63 {
64     double ergebnis;
65     double h = 0.000001;
66
67     wert = wert * M_PI/180;
68
69     ergebnis = (g1(wert + h) - g1(wert)) / h;
70
71     return ergebnis;
72 }
73
74 // Quelle: http://www.zeiner.at/c/index.html
75 int inverseMatrix(double a[2][4], double ainv[2][4], int n)
76 {
77     int zeile, spalte;
78     int schritt;
79     int pivotZeile;
80     bool fehler = false;
81     double multiplikationsfaktor;
82     const double genauigkeit = 0.00001;
83     double maximum;
84     bool pivot = true;
85

```

```

86 // Matrix zu einer Einheitsmatrix transformieren
87 for (zeile = 0; zeile < n; zeile++)
88 {
89     for (spalte = 0; spalte < n; spalte++)
90     {
91         a[zeile][spalte + n] = 0.0;
92         if (zeile == spalte) a[zeile][spalte + n] = 1.0;
93     }
94 }
95
96 // Eliminationsschritte
97 schritt = 0;
98 do
99 {
100     maximum = fabs(a[schritt][schritt]);
101     if (pivot)
102     {
103         pivotZeile = schritt;
104         for (zeile = schritt+1; zeile < n; zeile++)
105         {
106             if (fabs(a[zeile][schritt]) > maximum)
107             {
108                 maximum = fabs(a[zeile][schritt]);
109                 pivotZeile = zeile;
110             }
111         }
112     }
113     if (maximum < genauigkeit) break; // Fehler zu gross, nicht loesbar
114
115     if (pivot)
116     {
117         if (pivotZeile != schritt) // Zeilen tauschen
118         {
119             double h;
120             for (spalte = schritt; spalte < 2*n; spalte++)
121             {
122                 h = a[schritt][spalte];
123                 a[schritt][spalte] = a[pivotZeile][spalte];
124                 a[pivotZeile][spalte] = h;
125             }
126         }
127     }
128
129     // Eliminationszeile durch Pivot-Koeffizienten  $f = a[schritt][schritt]$  dividieren
130     multiplikationsfaktor = a[schritt][schritt];
131     for (spalte = schritt; spalte < 2 * n; spalte++)
132     {
133         a[schritt][spalte] = a[schritt][spalte] / multiplikationsfaktor;
134     }
135
136     // Nullen in Spalte "schritt" oberhalb und unterhalb der Diagonalen durch Addition
137     // der mit f multiplizierten Zeile "schritt" zur jeweiligen Zeile "zeile"
138     for (zeile = 0; zeile < n; zeile++)
139     {

```

```

140         if (zeile != schritt)
141         {
142             multiplikationsfaktor = -a[zeile][schritt];
143             for (spalte = schritt; spalte < 2 * n; spalte++)
144             {
145                 a[zeile][spalte] += multiplikationsfaktor * a[schritt][spalte];
146             }
147         }
148     }
149     schritt++;
150 } while (schritt < n);
151
152 if (maximum < genauigkeit) return 0; // singuläre Matrix
153
154 for (zeile = 0; zeile < n; zeile++)
155 {
156     for (spalte = 0; spalte < n; spalte++)
157     {
158         ainv[zeile][spalte] = a[zeile][spalte + n];
159     }
160 }
161 return 1;
162 }
163
164 // Funktionszeiger mit pnr (wonach soll abgeleitet werden)
165 double partielleAbleitung(double (*g)(double, double, double, double), double x1, double x2, double
166     x3, double x4, int pnr)
167 {
168     double ergebnis, tmp1, tmp2;
169     double h = 0.0000001;
170
171     switch(pnr)
172     {
173         case 0: tmp1 = g(x1+h, x2, x3, x4); break;
174         case 1: tmp1 = g(x1, x2+h, x3, x4); break;
175         case 2: tmp1 = g(x1, x2, x3+h, x4); break;
176         case 3: tmp1 = g(x1, x2, x3, x4+h); break;
177     }
178
179     tmp2 = g(x1, x2, x3, x4);
180     ergebnis = (tmp1 - tmp2) / h;
181     return ergebnis;
182 }
183 void jacobiMatrixIteration_2(double t1, double t2, double t3, double t4, double matrix[2][4])
184 {
185     int zeile, spalte;
186
187     for (zeile = 0; zeile < 2; zeile++)
188     {
189         for (spalte = 0; spalte < 2; spalte++)
190         {
191             if (zeile == 0) matrix[zeile][spalte] = partielleAbleitung(g2, t1, t2, t3, t4, spalte+1);
192             if (zeile == 1) matrix[zeile][spalte] = partielleAbleitung(g3, t1, t2, t3, t4, spalte+1);

```

```

193     }
194   }
195 }
196
197 void jacobiMatrixIteration_3(double t1, double t4, double t5, double t6, double matrix[2][4])
198 {
199   int zeile, spalte;
200
201   for (zeile = 0; zeile < 2; zeile++)
202   {
203     for (spalte = 0; spalte < 2; spalte++)
204     {
205       if (zeile == 0) matrix[zeile][spalte] = partielleAbleitung(g4, t1, t4, t5, t6, spalte+2);
206       if (zeile == 1) matrix[zeile][spalte] = partielleAbleitung(g5, t1, t4, t5, t6, spalte+2);
207     }
208   }
209 }
210
211 // Berechnung der Winkel 2–4 mittels Newton
212 void berechneWinkel()
213 {
214   double genauigkeit = 0.0001;
215   int iterationen_maximal = 50;
216   int iteration = 0;
217   double t1 = theta1;
218   double t2 = theta2;
219   double t3 = theta3;
220   double t4 = theta4;
221   double t5 = theta5;
222   double t6 = theta6;
223   double t2_alt, t3_alt, t4_alt, t5_alt, t6_alt;
224   double g1_wert, g1s_wert, g2_wert, g3_wert, g4_wert, g5_wert;
225   double winkelunterschiede;
226   double jacobi_matrix[2][4];
227   double jacobi_matrix_inv[2][4];
228   double tmp_vektor[2];
229
230   // Iteration von Theta4 bei Schnitt am Gelenkpaar
231   g1_wert = g1(t4);
232   g1s_wert = g1s(t4);
233
234   while (iteration < iterationen_maximal)
235   {
236     iteration++;
237     g1_wert = g1(t4);
238     g1s_wert = g1s(t4);
239     t4 = t4 - g1_wert / g1s_wert;
240     if (fabs(g1_wert) <= genauigkeit) break;
241   }
242
243   iteration = 0;
244   // Iteration von Theta2 und Theta3 bei Schnitt am Kugelgelenk
245   while (iteration < iterationen_maximal)
246   {

```

```

247     iteration++;
248     jacobiMatrixIteration_2(t1, t2, t3, t4, jacobi_matrix); // Jacobi-Matrix berechnen
249     inverseMatrix(jacobi_matrix, jacobi_matrix_inv, 2); // Jacobi-Matrix invertieren
250
251     // Funktionsvektoren
252     g2_wert = g2(t1, t2, t3, t4);
253     g3_wert = g3(t1, t2, t3, t4);
254
255     // Funktionsvektor * inverse Jacobi-Matrix
256     tmp_vektor[0] = jacobi_matrix_inv[0][0] * g2_wert + jacobi_matrix_inv[0][1] * g3_wert;
257     tmp_vektor[1] = jacobi_matrix_inv[1][0] * g2_wert + jacobi_matrix_inv[1][1] * g3_wert;
258
259     t2_alt = t2;
260     t3_alt = t3;
261
262     t2 = t2 - tmp_vektor[0];
263     t3 = t3 - tmp_vektor[1];
264
265     winkelunterschiede = fabs(t2 - t2_alt) + fabs(t3 - t3_alt);
266     if (winkelunterschiede <= genauigkeit) break;
267 }
268
269 // Iteration von Theta5 und Theta6 bei Schnitt am Drehgelenk
270 iteration = 0;
271 while (iteration < iterationen_maximal)
272 {
273     iteration++;
274
275     jacobiMatrixIteration_3(t1, t4, t5, t6, jacobi_matrix);
276     inverseMatrix(jacobi_matrix, jacobi_matrix_inv, 2);
277
278     // Funktionsvektoren
279     g4_wert = g4(t1, t4, t5, t6);
280     g5_wert = g5(t1, t4, t5, t6);
281
282     // Funktionsvektor * inverse Jacobi-Matrix
283     tmp_vektor[0] = jacobi_matrix_inv[0][0] * g4_wert + jacobi_matrix_inv[0][1] * g5_wert;
284     tmp_vektor[1] = jacobi_matrix_inv[1][0] * g4_wert + jacobi_matrix_inv[1][1] * g5_wert;
285
286     t5_alt = t5;
287     t6_alt = t6;
288
289     t5 = t5 - tmp_vektor[0];
290     t6 = t6 - tmp_vektor[1];
291
292     winkelunterschiede = fabs(t5 - t5_alt) + fabs(t6 - t6_alt);
293     if (winkelunterschiede <= genauigkeit) break;
294 }
295
296 // neue Winkel uebertragen
297 theta1 = t1;
298 theta2 = t2;
299 theta3 = t3;
300 theta4 = t4;

```

```

301     theta5 = t5;
302     theta6 = t6;
303 }

```

koerper.cpp:

```

1  #include <cmath>
2  #include <GL/gl.h>
3  #include <GL/glut.h>
4
5  void zylinder(float _laenge, float _radius, float _iterationen, char _seite)
6  {
7      float winkel = (2*M_PI) / _iterationen;
8      float ableitung = 1.0;
9
10     float x = (_seite=='x'?0.0:cos(winkel));
11     float y = (_seite=='x'?cos(winkel):sin(winkel));
12     float z = (_seite=='x'?sin(winkel):0.0);
13
14     glPushMatrix();
15     glBegin(GL_QUAD_STRIP);
16     glVertex3f(_radius * x, _radius * y, _radius * z);
17
18     switch (_seite)
19     {
20         case 'x': x = _laenge / _radius; break;
21         case 'z': z = _laenge / _radius; break;
22     }
23
24     glVertex3f(_radius * x, _radius * y, _radius * z);
25
26     for (int iteration = 1; iteration <= _iterationen + 1; iteration++)
27     {
28         x = (_seite=='x'?0.0:cos(winkel*iteration));
29         y = (_seite=='x'?cos(winkel*iteration):sin(winkel*iteration));
30         z = (_seite=='x'?sin(winkel*iteration):0.0);
31
32         if (_seite == 'x') glNormal3f(_radius * ableitung * y, _radius * y, _radius * z);
33         if (_seite == 'z') glNormal3f(_radius * x, _radius * y, _radius * ableitung * x);
34
35         glVertex3f(_radius * x, _radius * y, _radius * z);
36
37         switch (_seite)
38         {
39             case 'x': x = _laenge / _radius; break;
40             case 'z': z = _laenge / _radius; break;
41         }
42
43         if (_seite == 'x') glNormal3f(_radius * ableitung * y, _radius * y, _radius * z);
44         if (_seite == 'z') glNormal3f(_radius * x, _radius * y, _radius * ableitung * x);
45         glVertex3f(_radius * x, _radius * y, _radius * z);
46     }
47     glEnd();
48     glPopMatrix();

```

```

49 }
50
51 void achse(char _seite)
52 {
53     float _thickness = 5.0;
54     glPushMatrix();
55     zylinder(20.0, _thickness, 50, (_seite=='z'?'x':'z'));
56     glTranslatef((_seite=='x'?-_thickness*2:20.0), 0.0, (_seite=='z'?-_thickness*2:20.0));
57     zylinder(_thickness * 4, 2, 50, _seite);
58     glPushMatrix();
59     glTranslatef((_seite=='x'?2.0:0.0), 0.0, (_seite=='z'?2.0:0.0));
60     zylinder(_thickness * 2, 2, 50, (_seite=='z'? 'x': 'z'));
61     glPopMatrix();
62     glPushMatrix();
63     glTranslatef((_seite=='z'?_thickness * 2:0.0), 0.0, (_seite=='x'?_thickness * 2:0.0));
64     zylinder(_thickness * 4, 0.5, 50, _seite);
65     glPopMatrix();
66     glPushMatrix();
67     glTranslatef((_seite=='x'?_thickness * 4 - 2.0:0.0), 0.0, (_seite=='z'?_thickness * 4 - 2.0:0.0));
68     zylinder(_thickness * 2, 2, 50, (_seite=='z'? 'x': 'z'));
69     glPopMatrix();
70
71
72     glPopMatrix();
73 }

```

Abbildungsverzeichnis

2.1	Hauptbestandteile, Drehachsen und Drehsinn beim Verfahren eines Gelenkarmroboters am Beispiel KUKA KR 30–3/KR 60–3/KR 30 L16	6
2.2	Nachbildung des Gelenkarmroboters mit geraden Strecken als Starrkörper	7
2.3	$G_1=0, G_2=90, G_3=-90, G_4=0, G_5=0, G_6=0$	8
2.4	$G_1=-10, G_2=80, G_3=-100, G_4=-10, G_5=-10, G_6=-10$	9
2.5	$G_1=-10, G_2=70, G_3=-110, G_4=-20, G_5=-20, G_6=-20$	9
2.6	$G_1=-10, G_2=60, G_3=-120, G_4=-30, G_5=-30, G_6=-30$	9
2.7	$G_1=-10, G_2=50, G_3=-130, G_4=-40, G_5=-40, G_6=-40$	10
2.8	$G_1=-10, G_2=40, G_3=-140, G_4=-50, G_5=-50, G_6=-50$	10
3.1	Geometrie eines ebenen Gelenkvierecks in der Ausgangslage (Maßzahlen in K_0) .	15
3.2	ebenes Gelenkviereck, Figur 1	17
3.3	ebenes Gelenkviereck, Figur 2	18
3.4	ebenes Gelenkviereck, Figur 3	18
3.5	ebenes Gelenkviereck, Figur 4	18
3.6	ebenes Gelenkviereck, Figur 5	19
3.7	ebenes Gelenkviereck, Figur 6	19
4.1	räumliches Gelenkviereck	32
4.2	Körperschnitt bei einer räumlichen kinematischen Schleife	32
4.3	linke Teilkette	34
4.4	rechte Teilkette	34
4.5	kinematische Schleife	35