

ArmBot

Internes technisches Papier
19.09.2021 13:56

Jörg Roth
TH Nürnberg
90489 Nürnberg
Joerg.Roth@th-nuernberg.de

1	Die ArmBot-Umgebung	5
2	Die Ausführungsumgebung für Roboter-Programme	6
2.1	Überblick	6
2.2	Der Robot-Controller.....	7
2.3	Skills	8
3	Der ArmBot-Simulator	10
3.1	Einleitung.....	10
3.2	Die Umgebungsdatei.....	11
3.3	Die Benutzungsschnittstelle.....	17
3.4	Kommandozeilen-Parameter.....	18
4	Die Kinematik-Simulation	21
4.1	Das Roboter-Modell	21
4.2	Die physikalische Simulation	23
4.2.1	Bälle aus Kollisionen herausbewegen	24
4.2.2	Anpassen der Geschwindigkeitsvektoren von Bällen	25
4.2.3	Elastischer Stoß zwischen Bällen	25
4.3	Geschlossene kinematische Ketten	26
5	Inverse Kinematik	30
5.1	Einleitung.....	30
5.2	Geschlossene Lösung am Beispiel 'Simple Arm'	30
5.3	Inverse Kinematik mit der Jacobi-Matrix	32
6	Kommunikation und Ereignisse, Trigger.....	35
7	Details zu Verfahren und Berechnungen	36
7.1	Anwendung einer Roto-Translation.....	36
7.2	Berechnung einer Rotation zwischen zwei Punkten	38
8	Weitere technische Details	40
8.1	Java-Module	40
8.2	JavaFX	41
9	Offene Probleme, noch nicht realisierte Funktionen	42
10	Abschlussarbeiten und IT-Projekte.....	44
11	Weitere Referenzen	45
12	Index	46

Vorwort und Historie des Projektes

Das ArmBot-Projekt ist thematisch bei nicht-mobilen Manipulatoren angesiedelt. Entsprechende Roboter finden sich in Kontext der Industrie-Robotik und haben eine lange Tradition. Nach den erfolgreichen Roboter-Projekten Carbot und Bugbot ist es das dritte Roboter-Projekt, dessen Hauptkomponente eine Simulationsumgebung ist. Es gab aber schon vom Ansatz her grundsätzliche Unterschiede zu den anderen Projekten:

- Während Carbot und Bugbot autonome, *mobile* Roboter sind, ist der ArmBot ein stationärer Roboter(-arm).
- Es gibt nicht den *einen* ArmBot. Im Gegensatz zu den mobilen Robotern, gibt es ein Rahmenwerk für alle möglichen Roboterarm-Konfigurationen. Der Entwickler eines Roboterarms kann verschiedene Konstruktionen per Programm erzeugen. Carbot und Bugbot waren dagegen fest – es konnten bestenfalls Sensoren und Subsysteme abgeschaltet werden.
- Während Carbot und Bugbot auch als reale Roboter außerhalb der Simulation existieren, war der ArmBot direkt darauf angelegt, durch *keine* echte Hardware repräsentiert zu werden.

Der Hintergrund dieser Entscheidungen: während autonome, mobile Roboter aus der Sicht der Forschung sicher die interessanteren Roboter sind, gibt es doch einige grundlegende Fragestellungen, die sich am besten an den einarmigen Robotern behandeln lassen. Hierbei ist insbesondere der Problembereich der inversen Kinematik zu nennen: entsprechende Fragestellungen ergeben sich zwar auch bei der Steuerung der Beine eines Krabblers – in Reinform lässt sich das allerdings besonders gut bei Manipulatoren behandeln.

Im Mai 2020 begann die Entwicklung. Aus der Physik-Simulation des Bugbot-Projektes konnten einige Bestandteile kopiert werden, allerdings ergaben sich wesentliche Unterschiede. Der wesentliche: während der Bugbot im Wesentlichen die Umgebung wahrnimmt und sich in ihr bewegt, liegt der Schwerpunkt eines Manipulators darin, die Umgebung zu *manipulieren* (daher der Name). Daher musste die Simulation so angepasst werden, dass der Roboter zumindest einige Objekte greifen und bewegen kann. In Ansätzen gab es das schon in den Vorgängerprojekten – hier konnte der Roboter durch Anstoßen bewegliche Bälle über die Ebene bewegen. Dieser Art der Bewegung ist allerdings für einen Greifer zu restriktiv. Ein wesentlicher Anteil der Erweiterungen hatte daher auch zu Anfang, die Bälle dreidimensional beweglich zu machen. Im Juni konnte der Greifer dann auch schon einen Ball virtuell greifen, anheben, fallen lassen, wobei er am Boden abprallte und per Dämpfung einige Male gesprungen ist. Die Bälle waren dann auch mittelfristig die einzigen manipulierbaren Objekte der Umgebung.

Im Juli 2020 wurde ein weiterer Meilenstein genommen: die Kinematik-Simulation unterstützte ab diesem Zeitpunkt so genannte *geschlossene kinematische Ketten*. Diese sind für die Simulation nicht ganz unkritisch, da im allgemeinen Fall die Lage von Komponenten erst über ein Optimierungsverfahren bestimmbar ist. Die realisierte Lösung: für einige Muster gibt es geschlossene Lösungen. Diese Muster werden bei der Konstruktion des Roboters automatisch erkannt, nicht unterstützte Muster werden abgelehnt. Ein weiterer Meilenstein: es werden in einer Simulation direkt mehrere, u.U. verschiedene ArmBots simuliert. In den Carbot- und Bugbot-Umgebungen war dies nur möglich, indem mehrere Umgebungen per Netzwerk zusammengeschaltet wurden. Die ArmBot-Umgebung kann dies jetzt mit einer einzigen Umgebung.

Im Oktober wurde ein neuer beweglicher Objekttyp hinzugenommen: der Quader. Auf den ersten Blick ähnlich den Bällen, allerdings haben Quader eine Ausrichtung im Raum. Da-

her kann man jetzt auch Probleme simulieren, die sich aus einer bestimmten Greifrichtung ergeben. Objekte können auch vor dem Absetzen gedreht werden.

Auch wurden schon mehrere Berechnungsverfahren zur inversen Kinematik realisiert, insgesamt: geschlossene Formeln, Downhill-Simplex-Näherungsverfahren und das allseitsbekannte Verfahren über die Jacobi-Matrizen.

Im April 2021 wurden das Modulkonzept von Java ab Version 9 eingeführt. Damit konnte das Projekt einfacher zerlegt werden. Das ist besonders für Projekte in Lehrveranstaltungen und Abschlussarbeiten ein großer Vorteil, da viel feingranularer kontrolliert werden kann, was sichtbar ist.

Ein Wort zur Form: dieses Dokument erhebt nicht den Anspruch einer wissenschaftlichen Arbeit. Weder stilistisch noch formal soll es an die Qualität, beispielsweise einer Abschlussarbeit heranreichen. So werden häufig Konzepte und Realisierungsdetails gemischt. Das ist für den Zweck dieses Dokuments eine pragmatische Vorgehensweise, für Abschlussarbeiten sollte man ein Dokument aber besser strukturieren. Dieses Dokument sollte also weder als Vorlage für einen wissenschaftlichen Schreibstil verwendet werden, noch gilt es als Rechtfertigungsgrund für Abschlussarbeiter, es genauso "schlecht" zu machen.

1 Die ArmBot-Umgebung

Die ArmBot-Umgebung wurde entwickelt, um Manipulatoren ("Roboter-Arme") zu simulieren und Kontrollprogramme zu testen. Es gibt *keine* reale Hardware hierzu. Die Manipulatoren können vom Simulationsentwickler in weiten Grenzen konfiguriert werden, insbesondere die Anzahl und Typen der Motoren und die Geometrie der Anordnungen (z.B. wie lang sind die Segmente zwischen den Motoren). Im Moment werden zwei Motorentypen unterstützt:

- *Servo*: es wird ein Winkel vorgegeben und der Motor läuft zu diesem Winkel
- *Linearmotor*: es wird eine Länge vorgegeben und der Motor verlängert/verkürzt sich auf diese Länge

Abb. 1 (links) zeigt einen einfachen simulierten Manipulator.

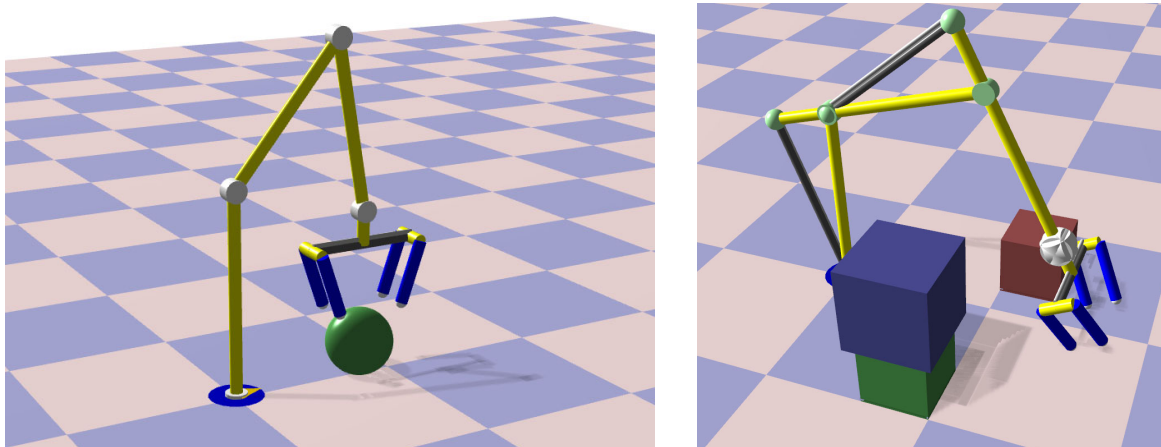


Abb. 1: ArmBots

Ein einfacher Aufbau besteht aus einer einzelnen kinematischen Kette: durch eine Sequenz von Motoren und Segmenten wird ein Aufbau von der Basis (*Body*) bis zum Greifer definiert. Als Abweichung von einfachen Ketten gibt es

- *Aufsplittung*: eine Kette kann auch geteilt werden (z.B. beim Greifer). Das erhöht die Komplexität der Simulation aber nicht, da die Lage jeden Endpunkts durch eine einzige Kette von der Basis bis zu diesem Punkt berechnet werden kann – es müssen nur die jeweiligen Roto-Translationen hintereinander ausgeführt werden.
- *Geschlossene Ketten*: hier werden Aufsplittungen wieder zusammengeführt. Damit kann die Simulation nicht mehr einfach die Endpositionen berechnen, da erst (in Echtzeit) entsprechende Gleichungssysteme gelöst werden müssen. Die Simulationsumgebung unterstützt derzeit nicht den allgemeinen Fall sondern nur einige Muster.

Als letztes: es können direkt mehrere Manipulatoren pro Simulation eingerichtet werden. Diese dürfen sogar unterschiedlich sein.

2 Die Ausführungsumgebung für Roboter-Programme

2.1 Überblick

Ein Entwickler hat verschiedene Möglichkeiten, die Umgebung zu kontrollieren. Die wichtigste: der *Robot-Controller* - dieser legt das Verhalten eines Roboters fest, also in welcher Sequenz er sich bewegt und wie er auf Ereignisse reagiert. Analog zu echten Manipulatoren, ist die Reaktion auf Ereignisse aber eher zweitrangig (im Gegensatz zu autonomen mobilen Robotern), sondern eher die Kontrolle der Motoren, um bestimmte Objekte der Umgebung zu manipulieren. Hierbei ist es zweitrangig, ob dieses Objekt tatsächlich an der erwarteten Position liegt, wenn es gegriffen wird.

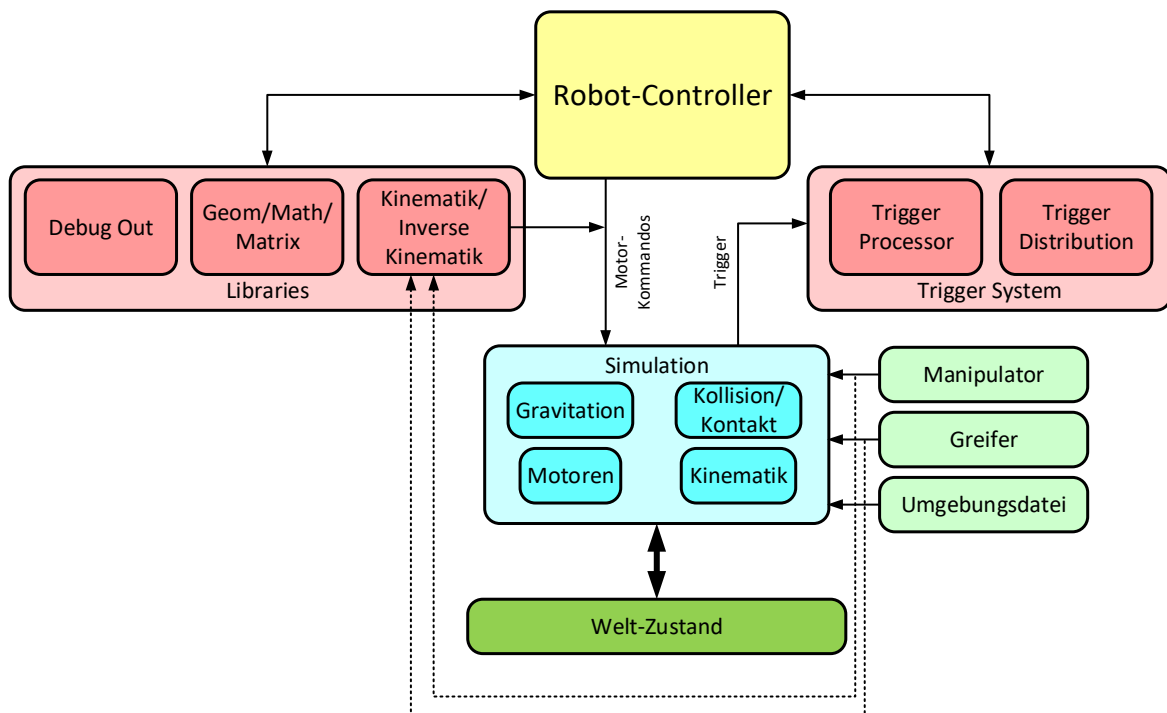


Abb. 2: Einbettung der Robot-Controller

Abb. 2 zeigt die Einbettung des Robot-Controllers in die Umgebung. Seine wichtigsten Schnittstellen:

- Mit Motor-Kommandos kann man den Roboter und damit letztlich die Umwelt beeinflussen, wenn Gegenständig bewegt werden.
- Über Trigger kann man Information über die Umwelt erhalten.
- Libraries unterstützen den Robot-Controller mit Dienstleistungen, insbesondere bei Kinematik-Berechnungen.

Es gibt weitere Möglichkeiten als Entwickler, Einfluss auf die Umgebung zu nehmen:

- Über die Umgebungsdatei wird die Umwelt, bestehend aus statischen Hindernissen und beweglichen Objekten definiert.
- Die Konstruktion des Manipulators und des Greifers kann als Programmcode hinterlegt werden.

Letztlich kann die Simulationsumgebung selbst angepasst werden. Dies ist aber nur in Ausnahmen sinnvoll.

2.2 Der Robot-Controller

Der Robot-Controller legt das eigentliche Verhalten des Roboters fest. Der Klassenname des Robot-Controllers wird der Laufzeitumgebung beim Start per Kommandozeilen-Parameter mitgeteilt. Der Entwickler realisiert einen Robot-Controller, indem eine Subklasse von `robotinterface.RobotController` implementiert wird.

Zur Realisierung muss der Robot-Controller einen parameterlosen Konstruktor anbieten (nur dieser wird vom Laufzeitsystem verwendet). Während der Instanziierung darf nur Code zur Ausführung kommen, der *einmalige* Einrichtungen betrifft. Insbesondere dürfen noch keine Aktionen wie Motorbewegungen ausgeführt werden. Später wird der Robot-Controller nur einmal instanziiert, der Benutzer kann ihn aber beliebig häufig erneut starten. Daher darf im Konstruktor kein Code hinterlegt werden, der bei jedem Start erneut ausgeführt werden soll.

Damit das Laufzeitsystem den Robot-Controller starten und stoppen kann, müssen die folgenden drei Methoden vom Entwickler definiert werden:

- **run () :**
Diese Methode kodiert die eigentliche Roboter-Aufgabe, z.B. ein Objekt zu greifen und abzulegen. Sie darf erst terminieren, wenn die Aufgabe erfüllt ist, oder von außen signalisiert wird, dass die Aufgabe vorzeitig beendet werden soll. Der Entwickler muss sicherstellen, dass die Methode nicht vorher beendet wird – ansonsten wertet das Laufzeitsystem die Aufgabe als erfüllt. Das wird im Simulator durch die Stati der Knöpfe *Start*, *Stop* oder *Pause* angezeigt.
Der Entwickler muss nicht selbst einen Bearbeitungsthread anlegen, es sei denn, es gibt innerhalb einer Aufgabe noch einmal eine nebenläufige Teilaufgabe. Um zu erkennen, dass eine Beendigung von außen angefordert wurde (z.B. über den Simulator-Knopf *Stop*), muss sich diese Methode beenden, wenn gilt `isRunning() == false`. Deshalb ist diese Bedingung regelmäßig zu testen, um ggfs. die Methode vorzeitig zu beenden.
- **stop () :**
Diese Methode wird nach Beendigung der Aufgabe aufgerufen, und auch, wenn die Beendigung manuell angefordert wurde. Sie muss auch zweimal hintereinander ausführbar sein, ohne dass Probleme entstehen. Diese Methode darf im Gegensatz zu **run ()** nicht blockieren. In dieser Methode werden beispielsweise alle Motoren gestoppt und alle internen Zustände auf "Anfang" gesetzt. Der Robot-Controller muss danach wieder mit **run ()** erneut gestartet werden können.
- **pause () :**
Diese Methode wird durch den *Pause*-Knopf zur Ausführung gebracht. Sie ist dem *Stop* sehr ähnlich, d.h. der Roboter wird in den Ruhezustand versetzt. Auch diese Methode darf nicht blockieren. Allerdings wird der interne Zustand nicht zurückgesetzt. Damit kann mit **run ()** die aktuelle Aufgabe fortgesetzt werden.

Die Idee hinter diesen drei Funktionen ist, dass man ähnliche Kontrollen wie bei einem Wiedergabegerät (z.B. für Videos) zur Verfügung hat: Man startet einen Lauf, pausiert eventuell, um die Situation zu analysieren, lässt weiterlaufen – das Ende tritt entweder von selbst ein oder man stoppt manuell, so dass man wieder von vorne starten kann.

Die Implementierung von **pause ()** ist nicht immer trivial. Einige Aufgaben sind nur schwer zu pausieren, da ja zumindest die aktuelle Bewegung gestoppt werden muss und vielleicht nicht mehr nahtlos fortgeführt werden kann. Sollte die Implementierung zu aufwändig sein, kann der Entwickler entscheiden, die **pause ()**-Methode nicht zu implementieren. In diesem Fall wird automatisch stattdessen **stop ()** aufgerufen.

Weitere Funktionen können durch den Robot-Controller optional implementiert werden:

- **String getDescription():**
gibt einen Text zurück, der die Funktion des Robot-Controllers spezifiziert. Dieser wird beim Starten der Umgebung ausgegeben.
- **boolean requiresConfiguration(), configure(String params):**
Der Robot-Controller kann eine Konfiguration verlangen. In dem Fall wird beim Starten des Laufzeitsystems getestet, ob eine Konfiguration angegeben wurde – diese wird entweder als Kommandozeilen-Parameter mitgegeben oder in der Umgebungsdatei hinterlegt. Liegt eine Konfiguration vor, wird diese dem Robot-Controller in Form einer Zeichenkette *einmalig* per **configure()** vor dem ersten Ausführen von **run()** mitgegeben. Der Robot-Controller definiert dabei den Aufbau der Zeichenkette; beispielsweise eine Liste von Zahlen, getrennt durch ';'. Entspricht der Aufbau nicht den Erwartungen, meldet er das Problem in Form einer Exception (z.B. **IllegalArgumentException**).

Als letztes kann ein Robot-Controller Trigger empfangen:

- **receiveTrigger(String triggerString):** Diese Methode wird immer aufgerufen, wenn von außen ein Trigger eintrifft (entweder durch einen anderen Robot-Controller oder durch die Umgebung. Der Robot-Controller kann darauf reagieren.

Über die lokale Variable **instanceNumber** kann der Robot-Controller erfragen, welche Roboter-Instanz er kontrolliert. Damit kann eine Umgebung mit mehreren Robotern eines identischen Typs generiert werden, bei denen jeder Roboter (kontrolliert durch dieselbe Klasse eines Robot-Controllers) etwas anderes tun.

2.3 Skills

Die Option **-skills** des Simulators kann Fähigkeiten (*Skills*) ein- und ausschalten. Als Beispiel: Obwohl ein Manipulator automatisch über die Fähigkeit verfügt, seine direkte Kinematik selbst zu berechnen, kann das ausgeschaltet werden. Damit kann man die Aufgabe dieser Berechnung auf den Robot-Controller übertragen.

Die Skills werden als Komma-getrennte Liste von Schlüsselworten definiert, denen jeweils ein "+" oder "-" vorangestellt wird, z.B.

```
-skills +mandirkin,-maninvkin
```

Hierbei bedeutet "+", dass diese Fähigkeit eingeschaltet wird, "-" steht für Ausschalten. Einige Skills bedingen sich gegenseitig. So darf **+maninvkin** nur gesetzt werden, wenn **+mandirkin** gesetzt ist.

Ausgeschaltete Skills führen dazu, dass bestimmte Funktionen eine **UnsupportedOperationException** auslösen. Ein Robot-Controller kann entweder explizit darauf reagieren (**catch...**), oder der Robot-Controller beendet sich beim Versuch des Aufrufs automatisch – die fragliche Exception ist *unchecked*, d.h. sie muss nicht ausgewertet werden, damit beendet sich aber komplette Funktion (z.B. die **run**-Methode) mit dieser Exception.

Tabelle 1 zeigt die Liste aller Skills.

Tabelle 1: Liste der Skills des Kommando-Programms **Command** sowie für den Simulator

Skill	Beschreibung	Bedingung	Default
mandirkin	Manipulatoren (manipulator -Instanz von Robotcontroller) erzeugen beim Hochfahren automatisch ein geeignetes DirectKinematics -Objekt. Dies kann verhindert werden – in dem Fall muss der Robot-Kontroller selbst eine Instanz eintragen.	keine	+
maninvkin	Manipulatoren (manipulator -Instanz von Robotcontroller) erzeugen beim Hochfahren automatisch ein geeignetes InverseKinematics -Objekt. Dies kann verhindert werden – in dem Fall muss der Robot-Kontroller selbst eine Instanz eintragen.	keine	+

3 Der ArmBot-Simulator

3.1 Einleitung

Das Thema Simulation wird in zwei Kapiteln behandelt. Die Mechanismen, die hinter der Kinematik stehen (z.B. wie wird der Einfluss der Gravitation berechnet), werden im Abschnitt 4 (Seite 21) behandelt. In dem aktuellen Abschnitt werden die Details der Simulation als Software-Komponente dargestellt.

Der ArmBot-Simulator ist eine Testumgebung für die Entwicklung von Robot-Controllern. Die Funktionen:

- Anzeigen und Steuern eines oder mehrerer virtuellen ArmBots. Simulation von Bewegungen;
- Einbindung einer virtuellen Umgebung (z.B. mit Hindernissen), in der sich der simulierte ArmBot befindet und die er beeinflussen kann;
- *Debug-Out*: zeilenorientierte Debug-Ausgaben (analog zu `System.out.println`) können vom Robot-Controller abgesetzt werden und werden in einem eigenen Konsolen-Fenster im Simulator angezeigt; zusätzlich in einer Textdatei gespeichert.
- *Commander*: Damit der Robot-Controller zur Laufzeit Befehle empfangen kann, gibt es ein Fenster mit einer Textzeile. Die eingegebenen Texte werden als Befehle an den Robot-Controller gesendet. Der Aufbau der Kommandos wird vom Robot-Controller definiert.
- *Trigger-Console*: Damit können die versendeten Trigger eingesehen und eigene Trig-

Zehn gute Gründe, einen Simulator einzusetzen

1. *Der Simulator ist sehr bequem.* Man kann die Tests überall machen, also auch dort, wo man keinen echten Roboter-test durchführen könnte (beispielsweise im Bus, während der Heimfahrt).

2. *Der echte Roboter ist wertvoll.* Dabei ist nicht alleine der Einkaufswert gemeint. Einige Ersatzteile können nicht leicht beschafft werden. Darüber hinaus ist der Aufwand für Zusammenbau und Installation zu betrachten.

3. *Es gibt zu wenige Roboter.* Hierbei denke ich vor allem an Lehrveranstaltungen mit vielen Gruppen, die an Projekten arbeiten möchten. Um die wenigen Roboter einsetzen zu können, müsste man über Zeitpläne jeder Gruppe eine "Roboterzeit" zuordnen.

4. *Die Benutzung des echten Roboters ist kompliziert.* Es ist meist kein fertiges Produkt, bei dem man den Anschalter drückt und alles läuft. Man muss für den Betrieb einiges beachten.

5. *Der echte Roboter ist für sich selbst eine Gefahr.* Wer schon einmal gesehen hat, wie durch ein fehlerhaftes Programm alle Servos synchron und schlagartig bis zur mechanischen Grenze verdreht werden, vergisst das nicht so schnell. Fehlerhafte Software kann real großen Schaden anrichten, im Simulator eher nicht.

6. *Der echte Roboter ist nicht ungefährlich.* Hierbei denke ich nicht an Killer-Roboter wie den Terminator. Aber alleine der Betrieb von starken Motoren fordert einen gewissen Respekt. Durch lange Hebel können große Geschwindigkeiten am Greifer entstehen.

7. *Der echte Roboter ist eigentlich nie betriebsbereit, wenn man ihn braucht.* Oft ist nicht klar, ob immer die neueste Betriebssoftware läuft, ob man sich noch an das aktuelle Kennwort für die Installation erinnert, und so weiter...

8. *Der echte Roboter kann vielleicht noch nicht alles, was er können sollte.* Häufig kostet eine bestimmte Fähigkeit in der Realisierung mehr Zeit als erwartet. Der Simulator kann aber "so tun", als ob alles schon funktioniert und man kann beginnen, aufbauende Verfahren zu realisieren. Wenn später alles funktioniert, kann man es zusammenbauen.

9. *Veränderungen des Aufbaus sind aufwändig.* Vielleicht ist man mit der Konfiguration unzufrieden, man wünscht sich einen neuen Greifer oder man möchte testweise die Armlänge verändern. Dazu muss man nicht gleich aufwändige Umbaumaßnahmen beginnen. Im Simulator kann man Modifikationen austesten, ohne dass man die echte Hardware anfassen muss.

10. *Der Aufbau einer Testumgebung ist in der Realität sehr aufwändig.* Man braucht viel den Platz, um alles aufzustellen. Im Simulator kostet jede Wand nur ein paar Zeilen Code.

ger versendet werden.

- *Status-Kontrolle*: Damit man ohne Neustart mehrmals Tests durchführen kann und auch einen Testlauf zur Analyse eines Problems unterbrechen kann, gibt es die Möglichkeit, einen Testlauf zu starten, zu stoppen und weiterzuführen.

Der Simulator simuliert die Physik in sehr rudimentärer Form. Es werden die klassischen Bewegungsgesetze angewendet, allerdings gibt es Vereinfachungen; so sind die Motoren unendlich stark und es gibt keine Verbiegungen oder elastische Verformungen. Darüber hinaus gibt es noch einige Inkonsistenzen, als Konsequenz einer echtzeitfähigen Simulation.

3.2 Die Umgebungsdatei

In der Umgebungsdatei kann man textuell eine Umgebung definieren, in der sich der ArmBot befindet. Die Umgebung besteht aus

- konstanten Definitionen (Position und Grenzen der Umgebung);
- Hindernissen: diese sind z.B. blockartige Objekte (**STEP**, **SLOPE**);
- beweglichen Objekten (derzeit **CUBOID** und **BALL**).

Im Folgenden wird die Syntax der Elemente im Einzelnen beschrieben. Positionen und Längenangaben werden in cm, Winkel in Grad angegeben.

Kommentare

- einzeilig:

```
// Kommentar
```

- mehrzeilig:

```
/* Mehrzeiliger Bereich
...
*/
```

Debugging-Namen

Mit

```
DEBUGNAME <name>
```

kann man Hindernissen einen Namen zuordnen, der beim Debugging der Physiksimulation angezeigt werden kann. Diese Direktive hat keine weitere Auswirkung auf die Simulation. Die Direktive bewirkt eine Zuordnung der direkt darauf folgenden Hindernis-Definition (z.B. **STEP**, **SLOPE**, auch **BALL**).

Nummer-Variablen

Mit

```
NUMBER <name> <wert>
```

kann man Nummer-Variablen anlegen. Variablen können an allen Stellen eingesetzt werden, an denen Fließkomma- und Integer-Werte in Direktiven vorkommen, z.B. <x> oder <y> in der **ROBOT**-Direktive. Solche Werte können auch durch einfache mathematische Ausdrücke zu Endwerten verarbeitet werden. Z.B.

```
NUMBER TARGET_X 100
NUMBER TARGET_Y 200
ROBOT0 TARGET_X*2+1 TARGET_Y-100 45
```

In einigen wenigen String-Parametern können auch mathematische Ausdrücke vorkommen, z.B. **CONFIG** (Seite 13). Hierzu muss allerdings der Anteil, der ersetzt werden soll, im String durch bestimmte Begrenzer markiert werden: **_I{...}_** für Integer-Werte, **_D{...}_** für Double-Werte. Z.B. ergibt **HALLO_I{1+1}_** den effektiven String **HALLO2**.

String-Variablen

Mit

```
STRING <name> <string>
```

kann man String-Variablen anlegen. Damit können beispielsweise Trigger-Nachrichten einmal definiert und an verschiedenen Stellen eingesetzt werden. Hierzu muss allerdings der Anteil, der durch die Definition ersetzt werden soll, im String durch den Begrenzer **_S{...}_** markiert werden, beispielsweise wie folgt:

```
STRING TRIG AN_OWN_TRIGGER
TRIGGER BALLENTERS 99 _S{TRIG}
```

Konstante Definitionen

- Roboter-Anzahl:

```
MULTIROBOT <anzahl>
```

Hiermit wird die Anzahl der simulierten Roboter (1...10) definiert. Diese Angabe muss in der Umgebungsdatei vor der ersten Direktive **CONTROLLER**, **MANIPULATOR**, **GRIPPER**, **ROBOT** oder **CONFIG** erscheinen. Erscheint sie überhaupt nicht, darf nur ein Roboter definiert werden.

- Startposition des ArmBots:

```
ROBOT <x> <y> <angle>
ROBOT<nr> <x> <y> <angle>
```

Der Winkel wird in Grad gemessen, positive Winkel *gegen* den Uhrzeigersinn, 0° gehen in *y*-Richtung, -90° in *x*-Richtung. *x*, *y* werden in cm gemessen. Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-robot**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Wird eine Roboter Nummer (<nr>) angegeben, so muss sie zwischen 0 und 9 liegen. Wird keine Nummer angegeben, so wird 0 angenommen.

- Grenzen der Umgebung in cm

```
BORDER <minx> <miny> <maxx> <maxy>
```

- Einfärben von Bodenfliesen

```
TILE <rangex> <rangey> <hexcol>
```

Die Bereiche werden in der Form *nr*, oder *von~bis* angegeben, z.B. **TILE 10 7~12 F0E0A0**. Die Nummern beziehen sich auf 50 cm-Kacheln, wobei 0, 0 die Kachel vorne links ist. Für Bereiche, die nicht über diese Direktive mit Farbinformationen belegt wurden, wird ein konstantes Schachbrettmuster verwendet.

- Definition der Robot-Controller-Klasse:

```
CONTROLLER <class>
CONTROLLER<nr> <class>
```

Angabe einer Robot-Controller-Klasse (eine Subklasse von **robot-interface.RobotController**). Hiermit wird das Verhalten des ArmBots gesteuert. Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-c**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Wird eine Roboternummer (<nr>) angegeben, so muss sie zwischen 0 und 9 liegen. Wird keine Nummer angegeben, so wird 0 angenommen.

- Definition der Robot-Controller-Konfiguration:

```
CONFIG <config>
CONFIG<nr> <config>
```

Diese Konfiguration wird einem Robot-Controller als Konfiguration mitgegeben. Der Aufbau hängt vom jeweiligen Robot-Controller ab. Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-cfg**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Wird eine Roboternummer (<nr>) angegeben, so muss sie zwischen 0 und 9 liegen. Wird keine Nummer angegeben, so wird 0 angenommen.

Der String <config> unterstützt eingebettete Integer-, Double- oder String-Werte durch die Angabe mathematischer oder String-Ausdrücke (Seite 11).

- Definition des Manipulators:

```
MANIPULATOR <class>
MANIPULATOR<nr> <class>
```

Angabe einer Manipulator-Klasse (eine Subklasse von **kin.sim.ManipulatorModel**). Hiermit wird der Roboter-Arm definiert (Geometrie, steuerbare Motoren etc.). Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-man**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Wird eine Roboternummer (<nr>) angegeben, so muss sie zwischen 0 und 9 liegen. Wird keine Nummer angegeben, so wird 0 angenommen.

- Definition des Greifers:

```
GRIPPER <class>
GRIPPER<nr> <class>
```

Angabe einer Greifer-Klasse (eine Subklasse von **kin.sim.GripperModel**). Hiermit wird der Roboter-Greifer definiert. Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-gri**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Wird eine Roboternummer (<nr>) angegeben,

so muss sie zwischen 0 und 9 liegen. Wird keine Nummer angegeben, so wird 0 angenommen.

- Definition des Trigger-Prozessors:

```
TRIGGERPROCESSOR <instnr> <class>
```

Angabe einer Trigger-Prozessor-Klasse (eine Subklasse von **robotinterface.TriggerProcessor**). Hiermit können Trigger-Nachrichten zu neuen Nachrichten verarbeitet werden. Die Nachrichten werden über die <instnr> empfangen.

Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-tp**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität.

- Definition der Trigger-Prozessor-Konfiguration:

```
TPCONFIG <config>
```

Diese Konfiguration wird dem Trigger-Prozessor als Konfiguration mitgegeben. Der Aufbau hängt vom jeweiligen Trigger-Prozessor ab. Diese Definition erfüllt denselben Zweck wie die Definition über die Kommandozeile mit **-tpcfg**. Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität.

Der String <config> unterstützt eingebettete Integer-, Double- oder String-Werte durch die Angabe mathematischer oder String-Ausdrücke (Seite 11).

Angabe von 2D-Formen

Diese werden nur im Rahmen einer anderen Definition z.B. **STEP**, verwendet, nie isoliert. Es wird ein Grundriss definiert, der durch ein konkretes Objekt in die dritte Dimension gezogen wird.

- Rechteck:

```
RECT <centrex> <centrey> <sizeX> <sizeY>
```

- Kreis:

```
CIRCLE <centrex> <centrey> <rad>
```

- Polygon:

```
POLY <n>  
<px1> <py1>  
...  
<pxn> <pyN>
```

Bodenbemalung

Man kann 2D-Bereich auf dem Boden einfärben. Diese werden nur angezeigt, haben weiter keine Bedeutung für die Simulation.

```
BOTTOMPAINT <hexcol> <alpha>  
<shapedef> // siehe oben, z.B. RECT, CIRCLE, POLY  
ENDSLICK
```

Stufen

Eine Stufe ist ein Hindernis, auf denen sich bewegliche Objekte befinden können, bzw. an den diese abprallen. Stufen haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
STEP <height> <hexcol>
<shapedef>      // siehe oben, z.B. RECT, CIRCLE, POLY
ENDSTEP
```

Schrägen

Eine Schräge hat dieselbe Eigenschaft wie eine Stufe. Der einzige Unterschied: die Deckelfläche ist schräg. Damit kann eine Steigung erzeugt werden und z.B. Bälle können runterrollen. Im Moment muss eine Schräge eine rechteckige Grundfläche haben. Die Steigung wird durch zwei Höhen festgelegt, wobei die erste Höhe zur Seite mit kleiner x- oder y-Koordinate gehört. Über eine Richtung (**x** oder **y**) wird festgelegt, in welcher Richtung sich die Höhe verändert. Schrägen haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
SLOPE <height1> <height2> <direction> <hexcol>
<shapedef>      // siehe oben, nur RECT
ENDSLOPE
```

Scheiben

Eine Scheibe hat dieselbe Eigenschaft wie eine Stufe. Der einzige Unterschied: die Bodenfläche liegt nicht auf Bodenhöhe. Damit können Tischplatten oder Regalfächer modelliert werden. Scheiben haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
SLICE <heightbottom> <heighttop> <hexcol>
<shapedef>      // siehe oben, z.B. RECT, CIRCLE, POLY
ENDSLICE
```

Keile

Keile sind Schrägen mit einer schrägen Bodenfläche. Im Moment muss ein Keil eine rechteckige Grundfläche haben. Die Steigungen werden jeweils durch zwei Höhen festgelegt, wobei die erste Höhe zur Seite mit kleiner x- oder y-Koordinate gehört. Über eine Richtung (**x** oder **y**) wird festgelegt, in welcher Richtung sich die Höhe verändert. Keile haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
WEDGE <upperheight1> <upperheight2>
      <lowerheight1> <lowerheight2>
      <direction> <hexcol>
<shapedef>      // siehe oben, nur RECT
ENDWEDGE
```

Förderbänder

Damit Bälle und Quader auf ein höheres Niveau gehoben oder horizontal transportiert werden können, gibt es Förderbänder. Über eine Richtung **<direction>** (**x** oder **y**) wird festgelegt, in welcher Richtung sich die Höhe verändert. Ein Ball, der auf dem Förderband liegt, wird mit der durch das Förderband definierten Geschwindigkeit (**<speed>** in cm/s) gradlinig bewegt. Die Geschwindigkeit kann auch negativ sein, wenn die Bewegungsrichtung **-x** oder **-y** ist. Mit **<tomiddlespeed>** kann man einem beförderten Objekt eine Drift in Richtung Bandmitte geben.

Förderbänder können laufen und stehen. Mit <initial> (**RUN** oder **STOP**) wird angegeben, wie es sich bei Simulationsstart verhält. Über <instnr> kann das Förderband die Trigger-Kommandos **RUN** oder **STOP** gesteuert werden. Die <instnr> kann auch -1 sein. Dann werden keine Trigger-Kommandos zugestellt.

```
CONVEYOR <instnr> <initial> <height1> <height2>
          <direction> <speed> <tomiddlespeed>
          <hexcol>
<shapedef> // siehe oben, nur RECT
ENDCONVEYOR
```

Kugeln

Kugeln können an beliebigen Positionen im Raum gesetzt werden, können also auch im Raum schweben. Ihre Position ist (im Gegensatz zu **BALL**) fest. Sie werden analytisch durch Zentrum und Radius beschrieben, besitzen also keine 2D-Formendefinition. Sie haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
SPHERE <centerx> <centery> <centerz> <rad> <hexcol>
```

Bälle

Bälle sind bewegliche Objekte. Bälle unterliegen einer komplexen Simulation dynamischer Effekte. So können sie durch den Roboter beschleunigt werden, prallen an Hindernissen ab und beschleunigen andere Bälle. Der Rollwiderstand wird durch einen Resistance-Wert angegeben (0.0-1.0), der angibt, um welchen Faktor sich eine einmal eingenommene Geschwindigkeit nach einer Sekunde verringert hat. Darüber hinaus gibt es einen Bounce-Wert, der angibt, wie sich die Geschwindigkeit beim Abprallen verringert. Bälle haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
BALL <centerx> <centery> <centerz> <rad>
      <resistance> <bounce> <hexcol>
```

Quader

Quader sind weitere bewegliche Objekte. Sie unterliegen einer komplexen Simulation dynamischer Effekte – im Gegensatz zu Bällen haben Sie eine Ausrichtung im Raum. Sie können herunterfallen und durch den Roboter gegriffen werden. Quader haben eine Farbe, die als 6-Ziffern-Hexwert angegeben wird.

```
CUBOID <centerx> <centery> <centerz>
        <size> <sizey> <sizez> <hexcol>
```

Ereignisse

Ereignisse (*Trigger*) werden von der Umgebung automatisch ausgelöst, wenn bestimmte Bedingungen eintreten. Hierbei gibt immer an: <instance> die Empfänger-Nummer (0..., oder -1 für alle) und <triggerstr> der String, der das Ereignis spezifiziert.

```
TRIGGER BALLENTERS <instance> <triggerstr> <x> <y> <rad>
                  <height> <col>
TRIGGER CUBOIDENTERS <instance> <triggerstr> <x> <y> <rad>
                  <height> <col>
TRIGGER BALLLEAVES <instance> <triggerstr> <x> <y> <rad>
                  <height> <col>
TRIGGER CUBOIDLEAVES <instance> <triggerstr> <x> <y> <rad>
```


<height> <col>

Diese Ereignisse werden ausgelöst, wenn ein beliebiger Ball/Cuboid einen zirkulären Bereich (2D) einnimmt oder verlässt. Hierbei wird das Zentrum des Balls/Cuboids gewertet. Die Höhe wird ausschließlich zu Darstellungszwecken ausgewertet und definiert nicht den sensitiven Bereich. Die Angabe <col> kann sein: * für alle Farben oder ein 6-Ziffern-Hexwert. Wenn eine Farbe angegeben wird, dann wird der Trigger nur ausgelöst, wenn der Ball eine Farbe hat, der maximal 10 RGB-Werte von der betreffenden Farbe entfernt ist.

TRIGGER BALLBARRIER <instance> <triggerstr> <x1> <y1> <x2> <y2>
<height> <col>

TRIGGER CUBOIDBARRIER <instance> <triggerstr> <x1> <y1>
<x2> <y2> <height> <col>

Dieses Ereignis wird ausgelöst, wenn ein Ball/Cuboid eine Linie von links nach rechts überschreitet, ähnlich einer Lichtschranke. Das Ereignis erfordert, dass der Mittelpunkt die Linie überschreitet, nicht etwa schon ein beliebiger Anteil des Balls/Cuboids. Die Seite links vs. rechts wird gemessen vom ersten Punkt (<x1>, <y1>) in Richtung zweiten Punkt (<x2>, <y2>). Auch hier wird die Höhe ausschließlich zu Darstellungszwecken ausgewertet und definiert nicht den sensitiven Bereich. Die Angabe <col> wird genauso ausgewertet wie bei den Triggern oben.

Der String <triggerstr> unterstützt eingebettete Integer-, Double- oder String-Werte durch die Angabe mathematischer oder String-Ausdrücke (Seite 11).

3.3 Die Benutzungsschnittstelle

Abb. 3 zeigt die Fenster der Simulationsanwendung.

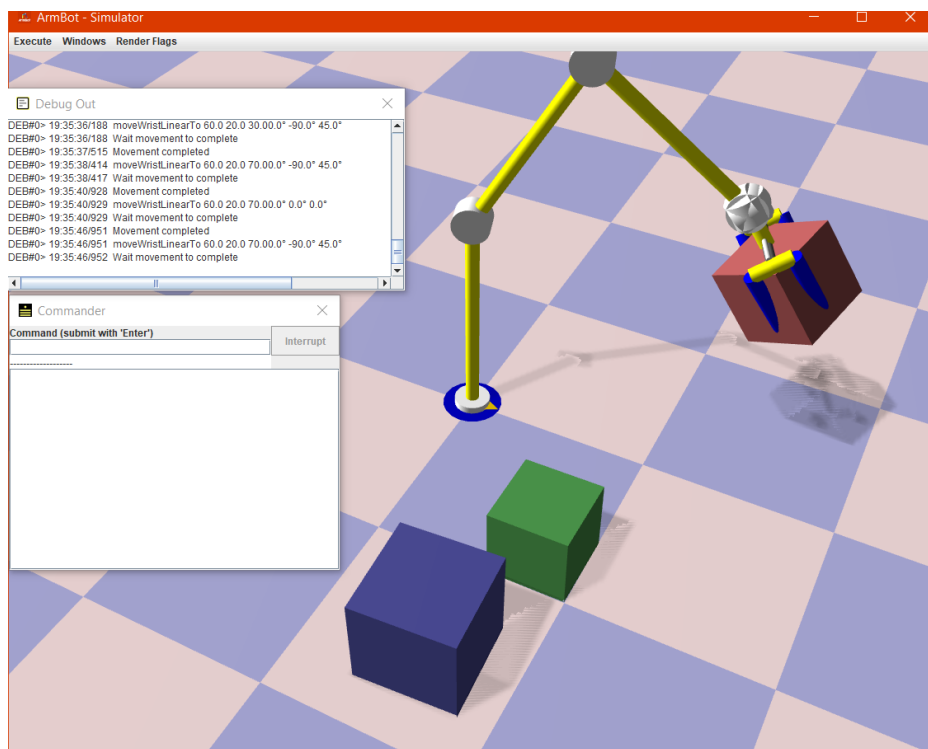






Abb. 3: Die Simulationsanwendung

Die Bereiche:

- Der größte Bereich stellt die Umgebung in 3D dar.
- Kleinere Fenster zeigen beispielsweise das Debug-Out-Fenster, das Commander-Fenster oder die Trigger-Console.
- Über Menüs erhält man Kontrollen für den Robot-Controller (Reset, Play, Stop, Pause), oder man kann die Zusatzfenster anzeigen.

Das Kontroll-Menü (*Execute*):

-  (Start): Start der Ausführung des Robot-Controllers;
-  (Stop): Stop der Ausführung des Robot-Controllers; Löschen des Zustandes;
-  (Pause): Stop der Ausführung des Robot-Controllers; Behalten des Zustandes;
-  (Reset): Zurücksetzen der Position auf den Anfang; Rücksetzen des Robot-Controllers.

Über die Maustasten kann man die Kamera-Position beeinflussen. Folgende Einstellungen sind möglich:

- Ziehen mit linker Maustaste: Drehen um den Fokuspunkt. Damit kann man den Roboter von allen Seiten betrachten.
- Ziehen mit linker Maus- und Shift-Taste: Bewegen des Fokuspunktes in x-y-Richtung. Damit kann man den Roboter aus der Bildmitte bewegen.
- Ziehen nach oben/unten mit rechter Maustaste: Ändern des Zooms; ziehen nach unten: weiter weg, nach oben: näher.

3.4 Kommandozeilen-Parameter

Die Simulationsumgebung kann mit folgenden Kommandozeilen-Parametern aufgerufen werden (Tabelle 2).

Tabelle 2: Kommandozeilen-Parameter der Simulationsumgebung

Kommandozeilen-Parameter	Beschreibung
-?	Ausgabe des Hilfstextes für die Kommandozeilen-Parameter
-e <environment>	Angabe der Umgebungsdatei
-man <manipulator> -man<nrs> <manipulator>	Angabe einer Manipulator-Klasse (eine Subklasse von kin.sim.ManipulatorModel). Hiermit wird der Roboter-Arm definiert (Geometrie, steuerbare Motoren etc.). Dieser Parameter erfüllt denselben Zweck wie die Direktive MANIPULATOR der Umgebungsdatei (Seite 13). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Variante mit <nrs> siehe unten.
-gri <gripper> -gri<nrs> <gripper>	Angabe einer Greifer-Klasse (eine Subklasse von kin.sim.GripperModel). Hiermit wird der Roboter-Greifer definiert. Dieser Parameter erfüllt denselben Zweck wie die Direktive GRIPPER der Umgebungsdatei (Seite 13). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Variante mit <nrs> siehe unten.

-ik <method>	Definiert die Methode zur Berechnung der inversen Kinematik: - pref : die bevorzugte Methode des Manipulators - closed : eine geschlossene, analytische Methode (wird nicht von allen Manipulatoren unterstützt) - down : Downhill-Optimierung - jac : Verfahren mit Hilfe der Jacobi-Matrix
-c <controller> -c<nrs> <controller>	Angabe einer Robot-Controller-Klasse (eine Subklasse von robotinterface.RobotController). Hiermit wird das Verhalten des ArmBots gesteuert. Dieser Parameter erfüllt denselben Zweck wie die Direktive CONTROLLER der Umgebungsdatei (Seite 13). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Variante mit <nr> siehe unten.
-tp <instnr>: <triggerproc>	Angabe einer Trigger-Prozessor-Klasse (eine Subklasse von robotinterface.TriggerProcessor) und der Instanz-Nummer zum Empfang von Triggern. Hiermit können Trigger-Nachrichten zu neuen Nachrichten verarbeitet werden. Dieser Parameter erfüllt denselben Zweck wie die Direktive TRIGGER-PROCESSOR der Umgebungsdatei (Seite 14). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität.
-tpcfg <config>	Angabe eines beliebigen Strings zur Konfiguration des Trigger-Prozessors. Dieser wird dem Trigger-Prozessor über den Aufruf configure() mitgeteilt. Der Aufbau des Strings wird vom Trigger-Prozessor definiert. Dieser Parameter erfüllt denselben Zweck wie die Direktive TPCONFIG der Umgebungsdatei (Seite 14).
-simst	Es wird regelmäßig der Ausführungspunkt aller Simulationsthreads in die Datei simthreads.out geschrieben. Sollte ein Thread durch einen Entwicklungsfehler blockiert sein (Endlosschleife oder blockierendes wait), so kann man das hierdurch entdecken.
-logstate	Der aktuelle Zustand des Robot-Controllers wird periodisch in die Datei state.out geschrieben. Hiermit kann man im Nachhinein analysieren, wo der Robot-Controller viel Rechenzeit verbraucht hat oder sich in einer Endlosschleife verfangen hat.
-logdebug	Debug-Out des Robot-Controllers wird in die Datei debug.out geschrieben.
-logsys	Systemmeldungen werden in die Datei syslog.out geschrieben.
-logall	Logge alles, was möglich ist. Das entspricht -logmss , -logstate , -logdebug , -logsys , -simst .
-logdir <dir>	Schreibe alle Log-Dateien in dieses Verzeichnis.
-lognames <str>	Für alle Log-Dateien (*.out) füge _<str> zum Stammnamen dazu (z.B. state_abc.out). Das ist sinnvoll, wenn man mehrere Instanzen des Simulators im selben Verzeichnis startet und die Log-Dateien unterscheiden möchte.

-cfg <config> -cfg<nr> <config>	Angabe eines beliebigen Strings zur Konfiguration des Robot-Controllers. Dieser wird dem Robot-Controller vor dem Start über den Aufruf configure() mitgeteilt. Der Aufbau des Strings wird vom Robot-Controller definiert. Dieser Parameter erfüllt denselben Zweck wie die Direktive CONFIG der Umgebungsdatei (Seite 13). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Variante mit <nr> siehe unten.
-robot <config> -robot<nr> <config>	Angabe eines Strings, der die Position des simulierten Robots zum Startzeitpunkt definiert. Dieser Parameter erfüllt denselben Zweck wie die Direktive ROBOT der Umgebungsdatei (Seite 12). Sollten beide Angaben gemacht worden sein, hat die Kommandozeile Priorität. Variante mit <nr> siehe unten.
-render <render>	Gibt an, über welchen Mechanismus die 3D-Umgebung dargestellt werden soll. <render> kann sein native : Hier wird das Bild über 2D-Polygone ohne 3D-Bibliothek dargestellt. fx : Hier wird das Bild über JavaFX 3D dargestellt. Auf älteren Plattformen gibt es eventuell keine Unterstützung dazu.
-renderms <ms>	Wenn gesetzt, wird die Wiederholungsrate der 3D-Darstellung auf alle <ms> Millisekunden begrenzt. Das ist sinnvoll, wenn die Rechenleistung für die Simulation begrenzt ist und man nicht auf eine flüssige grafische Ausgabe angewiesen ist.
-simtime <ratio>	Angabe der ablaufenden realen Zeit (ms), die 100 ms simulierte Zeit benötigen sollen. Bei 50 läuft die Simulation also doppelt so schnell wie die Realität.
-renderflags <flags>	Beeinflussung der 3D-Darstellung über gesetzte Bits. Hierzu gibt man eine Kette von T oder F an, in folgender Reihenfolge: <ul style="list-style-type: none"> Sollen die Servos, die sich gerade bewegen, besonders hervorgehoben werden? Sollen die Servos, die sich mit 95% oder mehr ihrer Maximalgeschwindigkeit bewegen, besonders hervorgehoben werden? Sollten die Trigger-Bereiche dargestellt werden?
-skills	Ein/Ausschalten von Fähigkeiten (siehe Abschnitt 2.3)

Hinweise zu **<nr>** und **<nrs>**: Diese Varianten können verwendet werden, um bestimmte Robots per Nummer zu referenzieren. Dazu muss die Umgebung mit **MULTIROBOT** (Seite 12) mehrere Roboter zulassen. Wird bei den Optionen keine Nummer oder kein Nummernbereich angegeben, wird "0" angenommen. Es gilt:

- <nr>**: hier steht eine einzelne Ziffer 0...9 (z.B. **-cfg5 <config>**). In diesem Fall gilt die Option für die spezielle Roboter-Instanz mit dieser Nummer. Wird die Nummer weggelassen, so wird 0 angenommen.
- <nrs>**: einige Optionen können für mehrere Roboter-Instanzen angewendet werden (z.B. **-c**). Damit man diese nicht wiederholen muss, erlaubt der Zusatz **<nrs>** direkt mehrere Instanzen anzusprechen: daher kann es sein: einzelne Nummer 0...9 (z.B. **-c5**), Bereich von Nummern, getrennt mit '-' (z.B. **-c1-3**) oder Nummern einzeln aufgeführt (z.B. **-c2357**). Wird nichts angegeben, so wird 0-9 angenommen.

4 Die Kinematik-Simulation

4.1 Das Roboter-Modell

Der Roboter wird als Modell hinterlegt. Das Modell legt dabei die Geometrie und die Verbindungen untereinander fest. Die einzelnen Teile können baumartig verknüpft werden – ausgehend vom *Body* können Segmente und Motoren angeknüpft werden. Hierbei sind Verzweigungen (*TConnectors*) möglich.

Die Richtung zum Körper wird *fix-side*, die Richtung zum Arm-Endpunkt wird *tail-side* bezeichnet. Darüber hinaus gibt es die Komponenten-Rollen *Fix-Side-Part* und *Tail-Side-Part*. Diese Rollen sind relevant, wenn man zwei Komponenten verbindet. Wenn ein bestimmter Typ auf der Seite in Richtung *fix-side* angebracht werden kann, ist es ein *Fix-Side-Part*, analog für *tail-side*. Ein Komponententyp kann beide Rollen einnehmen.

Eine Verbindung zwischen zwei Komponenten wird grundsätzlich über einen *Connectionpoint* gebildet. Ein *Connectionpoint* definiert vor allem die lokale Roto-Translation des *Tail-Side-Parts* gegenüber dem *Fix-Side-Part*. Damit definiert der *Connectionpoint*

- wo ist das *Tail-Side-Part* angebracht (Translation),
- wie ist es gedreht (Rotation).

Es gibt folgende Komponententypen:

- *Body* (nur *Fix-Side-Part*): das Wurzel-Element aller Roboter-Komponenten. Es gibt genau einen *Body* – hiervon verzweigt der Arm ab.
- *Segment*, *TouchSegment* (*Fix-Side-Part* und *Tail-Side-Part*): ein gerades Stück. Die Liniengeometrie des Segments wird durch den lokalen Nullpunkt bis zu einem einzigen *Connectionpoint* definiert.
- *TConnector* (*Fix-Side-Part* und *Tail-Side-Part*): Aufsplitten in mehrere *Tail-Side-Parts*.
- *Servo* (*Fix-Side-Part* und *Tail-Side-Part*): ein punkartiges Objekt, mit einem *Connectionpoint* im lokalen Nullpunkt. Die Rotation des *Connectionpoints* kann programmatisch beeinflusst werden.
- *LinearMotor* (*Fix-Side-Part* und *Tail-Side-Part*): ein linienartiges Objekt, mit einem beweglichen *Connectionpoint*. Die Länge bis zum *Connectionpoint* kann programmatisch beeinflusst werden.
- *TouchPoint*, *TerminationPoint* (nur *Tail-Side-Part*): ein punkartiges Objekt am Ende einer Kette.

Man kann zwei Komponenten davon herausgreifen: *TouchSegment* und *TouchPoint* (so genannte *Touchable*) sind Komponenten, die die Objekte der Umgebung beeinflussen können, d.h. Objekte prallen davon ab oder können dadurch verschoben werden.

Ein konkreter Roboter wird durch zwei Layer definiert (Abb. 4):

- *Model Layer*: hier wird das statische Modell definiert, also die statischen geometrischen Eigenschaften. Insbesondere sind die geometrischen Eigenschaften nur im lokalen Koordinatensystem in Relation zum *Connectionpoint* in Richtung *fix-side* gespeichert.
- *Dynamic Layer*: hier wird das dynamische Modell definiert. Das speichert gegenüber dem statischen Modell beispielsweise die konkrete Lage im Raum und konkrete Servo-Winkel.

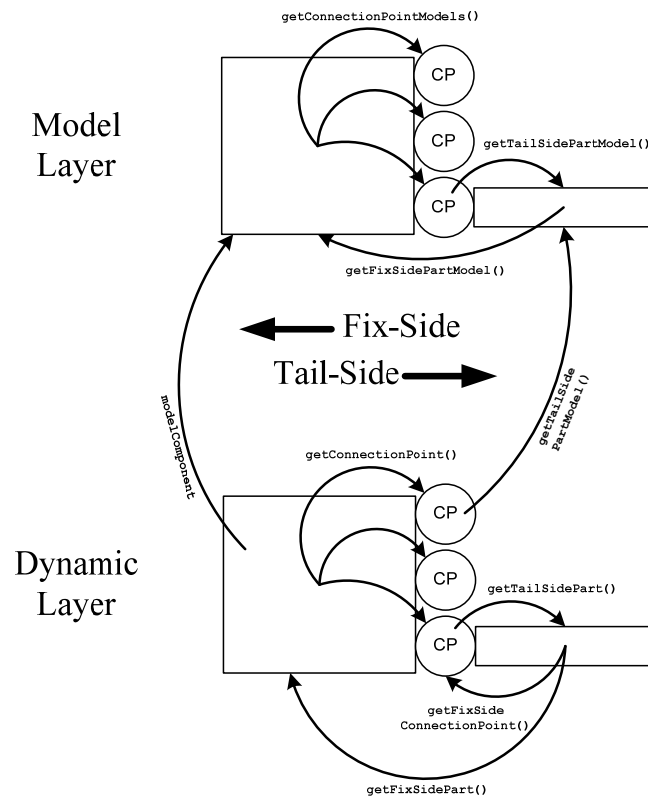


Abb. 4: Zusammenhang zwischen statischem und dynamischem Modell

Das statische Modell wird vom Entwickler als einziges explizit definiert. Das dynamische Modell wird zur Laufzeit vom statischen Modell abgeleitet und ist die eigentlich Instanz, über die simuliert wird. Hierbei kennt das dynamische Modell die zugehörigen statischen Komponenten. Bei allen Anfragen zu statischen Daten wird die Anfrage an die statische Komponente durchgereicht.

Zur Laufzeit wird noch ein dritter Layer benötigt, der *Render Layer* (Abb. 5).

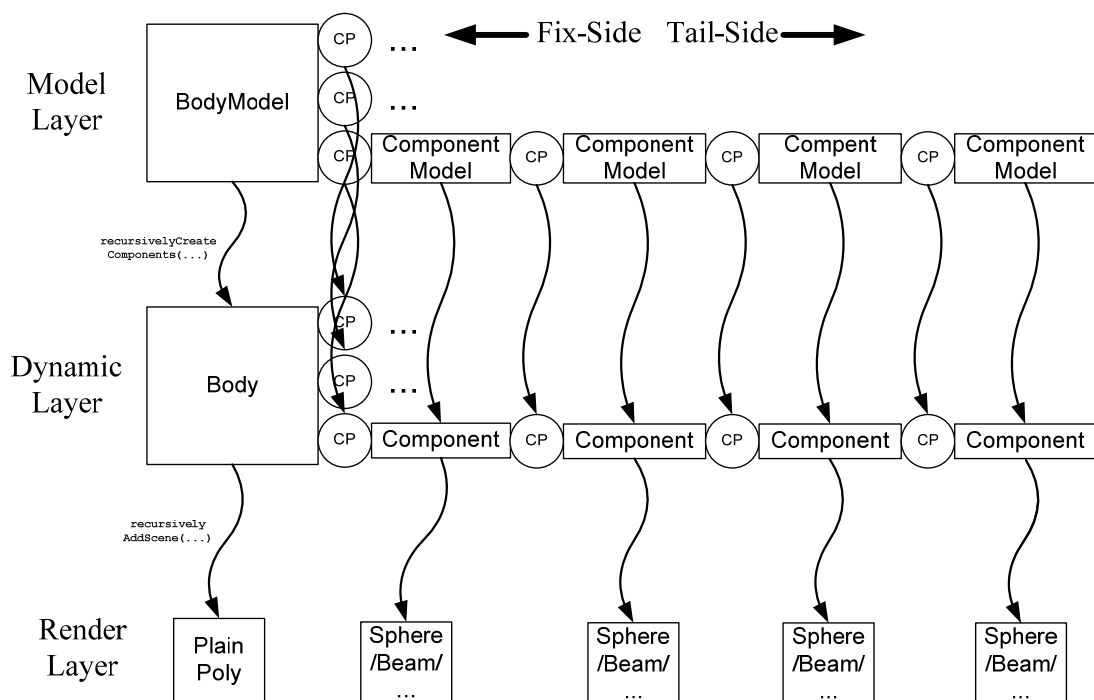


Abb. 5: Die drei Ebenen eines Modells

Dieser Layer wird ausschließlich zur Visualisierung benötigt. Die Komponenten des Render Layers orientieren sich an darstellbaren 3D-Komponenten (**RenderObj** bzw. **RenderObjFX**) und nicht mehr an dem eigentlichen kinematischen Modell.

4.2 Die physikalische Simulation

Abb. 6 zeigt die Schritte der Simulation.

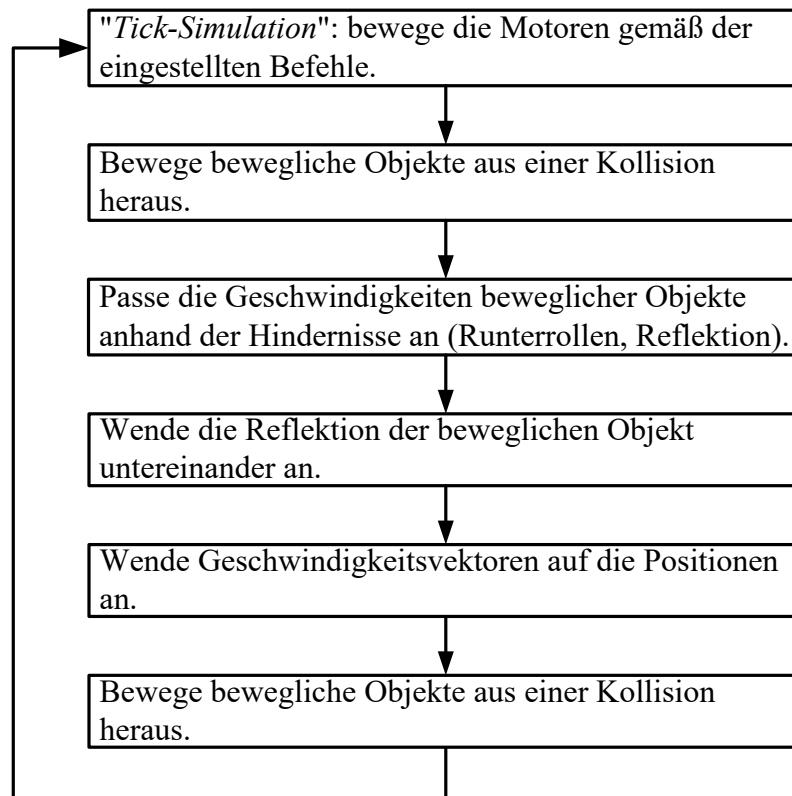


Abb. 6: Die Schritte der Simulation

Die Funktion "Bewege bewegliche Objekte aus einer Kollision heraus" wird tatsächlich zweimal aufgerufen. Der Grund: zu anfangs könnte eine neue Kollision entstanden sein, weil sich der Roboter bewegt hat. Erst auf der Basis der "bereinigten" Situation sollen weitere Schritte durchgeführt werden. Am Ende können durch die Anwendung der Geschwindigkeiten erneut Kollisionen auftreten. Da am Ende des Zyklus die Simulation bis zum nächsten Simulationsschritt stillsteht, soll auch hier eine bereinigte Situation hergestellt werden.

Die Simulation beweglicher Objekte ist eine wesentliche Funktion. Der Roboter-Arm soll bewegliche Objekte beeinflussen können, z.B.

- Anheben mit dem Greifer,
- Anschieben zum Rollen.

Im ersten Ansatz stehen nur Bälle und Quader (*Cuboid*) zur Auswahl. Bälle haben den Vorteil, dass die Lage nur durch eine Position, nicht zusätzlich durch eine Rotation definiert wird. Darüber hinaus ist der Test auf Kollision sowie die Berechnung der Reflektionsregeln sehr einfach. Quader können kippen aber nicht rollen oder nicht reflektiert werden. Werden sie beispielsweise von einem Ball getroffen, so reagiert der Ball so als ob das Hindernis eine Wand wäre.

Bei Bällen gibt es drei wesentliche Teileffekte im Rahmen der Simulation:

- Ein Ball schneidet sich in einem Touchable-Objekt des Roboters. Das passiert, wenn sich im letzten Zyklus der Roboter bewegt hat und ein Objektpunkt des Roboters in einen Ball bewegt wurde. Der häufigste Fall ist dabei, wenn der Greifer einen Ball anhebt. Die Konsequenz ist, dass die Bälle sich aus der Kollision heraus bewegen müssen.
- Der Ball berührt ein festes Hindernis. Das passiert, wenn ein Ball gegen ein Hindernis rollt oder auf ein Hindernis fällt (auch auf den Boden). Die Konsequenz ist, dass der Geschwindigkeitsvektor des Balls anhand der Berührung angepasst werden muss.
- Der Ball berührt einen anderen Ball. Das passiert, wenn ein Ball gegen einen anderen Ball rollt oder auf einen fällt. Die Konsequenz ist, dass der Geschwindigkeitsvektor beider Bälle anhand der Berührung angepasst werden muss.

Bei Quadern gibt es nur den folgenden Teileffekt:

- Ein Quader schneidet sich in einem Touchable-Objekt des Roboters. Das passiert, wenn sich im letzten Zyklus der Roboter bewegt hat und ein Objektpunkt des Roboters in einen Quader bewegt wurde. Der häufigste Fall ist dabei, wenn der Greifer einen Quader anhebt. Hier werden zwei Varianten unterschieden: 1. Es gibt mehrere Greifer, die den Quader berühren: in diesem Fall wird eine analoge Behandlung wie beim Ball durchgeführt. 2. Es ist nur ein Greifer involviert: in diesem Fall wird der Quader als Erweiterung der kinematischen Kette angesehen und führt alle Bewegungen des Greifers aus.

Die Effekte für Bälle werden am Folgenden beschrieben.

4.2.1 Bälle aus Kollisionen herausbewegen

Die Kollision mit Touchable-Objekten des Roboters und mit festen Hindernissen wird gleich behandelt. Liegt der Ball so, dass ein Hindernispunkt sich in ihm befindet, muss die Simulation den Ball so bewegen, dass nur noch eine Berührung vorliegt.

Ein erster Ansatz, bei dem mehrere Hindernispunkte auf einmal berücksichtigt wurden, wurde verworfen. Zur Erklärung: wenn der Ball beispielsweise drei Hindernisse berührt, kann man durch den Kugelschnitt um die drei Hindernisse den Ball-Mittelpunkt so berechnen, dass der Ball alle drei Hindernisse exakt berührt. Das führte einerseits zu einer Reihe von Fällen, andererseits konnte der Fall "2 Hindernispunkte" nie richtig geklärt werden.

Deshalb wird im Regelfall der nächste Hindernispunkt gewertet und der Ball so über die Flächennormale des Hindernisses wegbewegt, dass der Ball das Hindernis nur noch berührt (Abb. 7 links). Da diese Bewegung zu neuen Hindernissen führen kann, wird das mehrfach angewendet. Alle bei den Iterationen auftretenden Hindernispunkte werden aufgesammelt und der Anpassung der Geschwindigkeitsvektoren zugeführt.

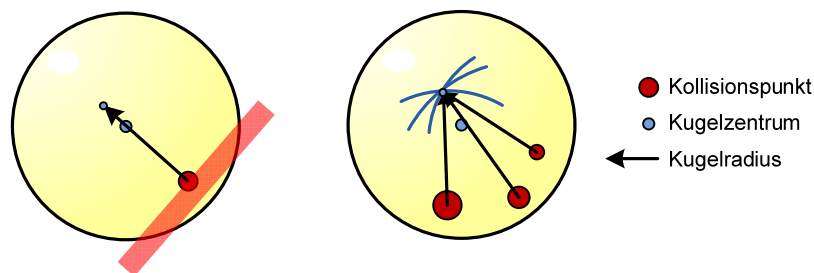


Abb. 7: Fälle beim Herausbewegen eines Balls aus Kollisionen

Dieser Ansatz funktioniert in einem Fall jedoch nur schlecht: wenn ein Greifer einen Ball anhebt. Hier gibt es viele Hindernispunkte (die am Greifersegment) und jeden einzeln zu

betrachten führt zu ruckeln. Darüber hinaus muss beim Greifen als Sonderfall die Ballgeschwindigkeit auf 0 gesetzt werden. Daher wird für den Fall

- mindestens drei Hindernispunkte *und*
- mindestens drei Touchpoint als Hindernis

gesondert behandelt: es werden die drei zum Ball-Mittelpunkt nächstgelegene Hindernispunkte betrachtet und die Kugeloberflächen um dieser herum geschnitten (Abb. 7 rechts). Das ergibt zwei Schnittpunkt – der dem Ball-Mittelpunkt nächstgelegene Schnittpunkt wird er neue Ballmittelpunkt. Darüber hinaus wird die Eigengeschwindigkeit auf 0 gesetzt.

4.2.2 Anpassen der Geschwindigkeitsvektoren von Bällen

Der Geschwindigkeitsvektor eines beweglichen Objektes wird auf zwei Arten beeinflusst:

- In negative z-Richtung wirkt die Erdbeschleunigung, d.h. pro Simulationsschritt wird ein entsprechender konstanter Geschwindigkeitsanteil addiert.
- Berührt ein Ball eine Fläche, so wird der Geschwindigkeitsanteil, der in die Fläche geht, invertiert (Reflektion) und der Anteil, der parallel zur Fläche geht bleibt erhalten ("Runterrollen"). Der Anteil der Reflektion wird darüber hinaus gedämpft, d.h. ein Ball springt nach einer Reflektion weniger hoch zurück.

Abb. 8 zeigt die Konstruktion des Geschwindigkeitsvektors.

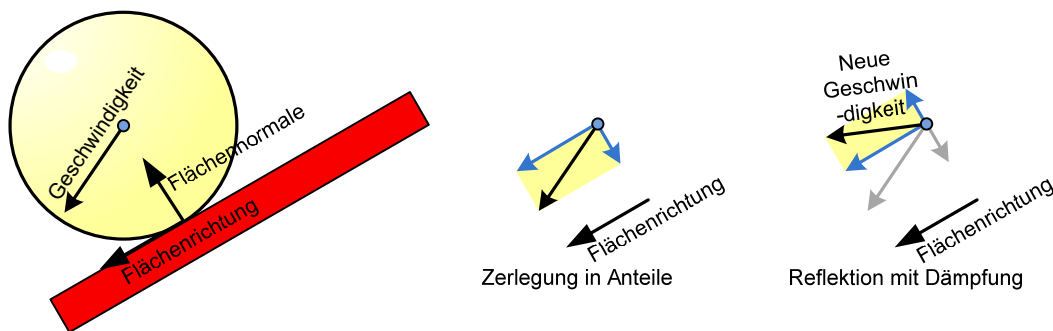


Abb. 8: Anpassung der Geschwindigkeit

Ein Ball erhält über diese Konstruktion eine neue Geschwindigkeit. Sollte sie mehrere Hindernisse oder Touchables in einem Takt berühren, so werden diese einzeln hintereinander gemäß dieser Konstruktion angewendet. Dem liegt folgendes Model zugrunde: sollten die Hindernisse kurz hintereinander berührt werden, so würde der Ball zweimal (wie beim Billard) reflektiert werden. Der gleichzeitige Kontakt ist dabei nur der Grenzwert von hintereinander angewendeten Zeitpunkten.

4.2.3 Elastischer Stoß zwischen Bällen

Eine Sonderbehandlung erhalten zusammenstoßende Bälle. Der Grund: bei allen anderen Kollisionen hat nur der Ball eine endliche Masse und verändert ihre Bewegung. Das andere Objekt ändert seine eigene Geschwindigkeit nicht.

Bei zwei Bällen ist das anders: jeder Ball verändert seinen Bewegungsvektor aufgrund der Kollision. Physikalisch steckt dahinter die Theorie des elastischen Stoßes: es wird angenommen, dass beim Aufprall keine Energie verloren geht. Daher muss vor und nach dem Stoß die Energie gleich bleiben. Dasselbe gilt für den Impuls.

Bei Bällen kann die Fragestellung relativ einfach aufgelöst werden (Abb. 9).

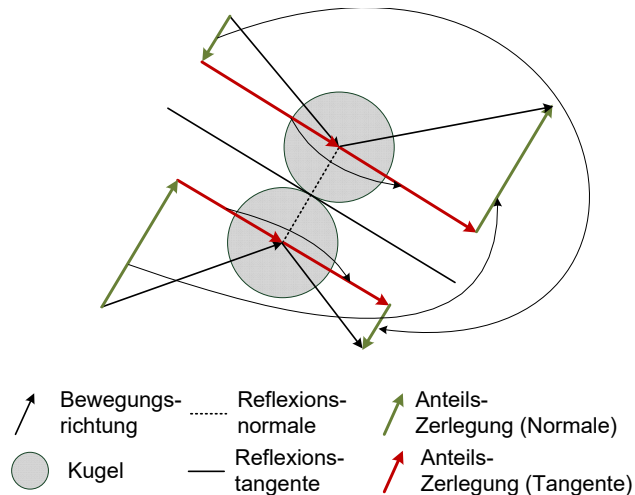


Abb. 9: Geschwindigkeiten vor und nach einem elastischen Stoß

Die Bälle treffen sich in einem Punkt. Die Verbindung der Mittelpunkte ergeben die Reflexionsnormale, rechtwinklig davon liegt die Reflexionstangente.

Jeder Bewegungsvektor wird jetzt in die entsprechenden Anteile zerlegt. Für die gilt:

- Der jeweilige Ball behält den Anteil, der parallel zur Reflexionstangente liegt (rot in Abb. 9).
- Jeder Ball übernimmt den Anteil des anderen Balls, der parallel zur Reflexionsnormale liegt (grün in Abb. 9).

Damit können bei einem Zusammenstoß die neuen Geschwindigkeitsvektoren leicht berechnet werden. Die Geschwindigkeitsvektoren müssen nur durch eine orthogonale Projektion auf die zwei Achsen verteilt werden.

4.3 Geschlossene kinematische Ketten

Das Modell in Abschnitt 4.1 beschreibt so genannte offene kinematische Ketten. D.h. ausgehend vom Start (Body) kann man alle Winkel und Längen "aufsummieren" um zur Greifer-Position zu gelangen. Konkret wird durch jede Komponente eine Roto-Translation definiert. Man startet bei der lokalen Greifer-Position und wendet die Roto-Translation der Komponente, an der der Greifer montiert ist. Dann wendet man die nächstgelegene Roto-Translation in Richtung Start an usw. Kennt man schließlich die globale Position des Starts, dann kann man die globale Pose des Greifers berechnen.

Diese Arte der kinematischen Kette wird 'offen' bezeichnet, da man eine Baum-Struktur ohne Schleifen definiert. Die Berechnung der Endposition ist alleine durch die Anwendung der Kette der Roto-Translationen ausgehend vom Greifer in Richtung Körper möglich. Das gilt auch für dynamische Komponenten (Servo oder Linearmotor), da diese zum Zeitpunkt der Berechnung eine definierte Größe (Servo-Winkel oder lineare Länge) einnehmen.

Im Gegensatz dazu gibt es geschlossene kinematische Ketten. Bei diesen werden bestimmte Punkte zusammengeführt. Geschlossene kinematische Ketten funktionieren nur mit passiven Komponenten – das sind Elemente, die ihre Länge oder Winkel anhand von außen angelegten Kräften so ausrichten, dass die Endpunkte der jeweilige Teilketten sich im gewünschten Punkt treffen.

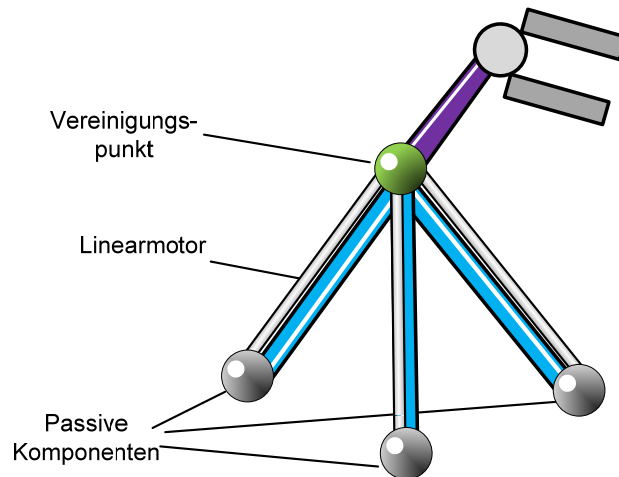


Abb. 10: Beispiel einer geschlossenen kinematischen Kette

Abb. 10 zeigt ein Beispiel: es gibt drei passive Punkte, deren Position bekannt sein muss. Drei Linearmotoren werden in einem Vereinigungspunkt zusammengeführt. Werden jetzt die Längen der Linearmotoren vorgegeben, ergeben sich eindeutige Roto-Translationen in den passiven Komponenten, damit die kinematische Bedingung erfüllt wird. Um eine geschlossene Kette im statischen Modell auszudrücken, benötigen wir ein Ausdrucksmittel.

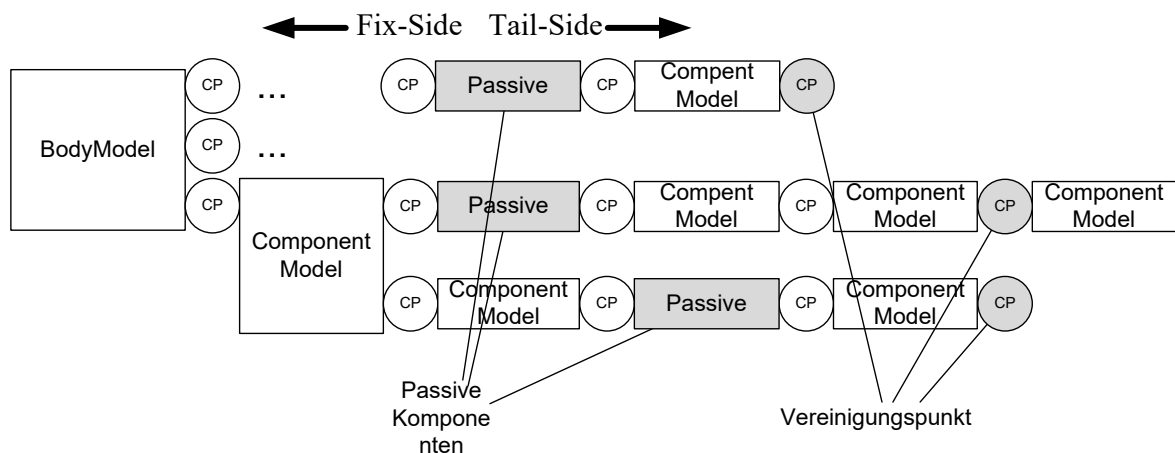


Abb. 11: Erweiterung des Modells um geschlossene Ketten auszudrücken

Abb. 11 zeigt den Ansatz: eine geschlossene Kette wird erst einmal durch mehrere offene Ketten definiert. Im Beispiel ergeben sich beispielsweise drei Ketten, die entweder durch unterschiedliche Startpunkte am Body oder durch Komponenten entstehen, die die Kette aufsplitten (z.B. TConnector).

Man hinterlegt jetzt im Model ein Objekt, das die drei passiven Komponenten, sowie die drei Connectionpoints verwaltet. Hierbei sollen die passiven Komponenten ihre jeweiligen Größen (z.B. Winkel) so einstellen, dass die Connectionpoints dieselbe Position einnehmen.

Die Berechnung der Größen der passiven Komponenten ist nicht trivial. Im Normalfall können diese nicht über Gleichungssysteme ausgedrückt werden, die man über geschlossene Verfahren berechnen kann. Das würde zu einem Optimierungsproblem führen, das versucht, den Abstand der zu vereinheitlichenden Positionen zu minimieren. Da die Berechnung aber im "Tick" der Simulation erforderlich ist, wäre ein Optimierungsverfahren kritisch. In der aktuellen Fassung werden daher nur bestimmte Muster geschlossener Ketten unterstützt. Beispielsweise kann die Konstruktion in Abb. 10 tatsächlich über einen einfachen Ansatz modelliert werden: es müssen drei Kugeloberflächen geschnitten werden.

Beim Erstellen der statischen Kette wird ermittelt, ob es sich um ein unterstütztes Muster handelt. Die eigentliche Berechnung findet im Zuge der Ausführung von `Component.recursivelySetRotoTranslationWorld()` durchgeführt.

Das jeweilige passive Element beschränkt die Lage der daran aufgehängten kinematischen Kette und beschränkt daher die Position des Vereinigungspunktes. Es werden folgende passiven Elemente unterstützt:

- Alle Rotationen (**FULLROT**) – das "Kugelgelenk": die angehängte Kette kann über beliebige Rotation gedreht werden. Genauer: die Rotation wird aus zwei Rotationen aufgebaut (erst in der x - y -Ebene, also um die z -Achse, dann in der x - z -Ebene, also um die y -Achse).
- 1-Achse-Rotationen (**AXISROT**) – das "Türscharnier": die angehängte Kette kann über eine Rotation gedreht werden (in der x - y -Ebene, also um die z -Achse).
- Alle Längen (**LINEAR**) – die "Teleskopstange": die angehängte Kette kann über eine Distanz verschoben werden (in Richtung der x -Achse).

Je nach Anzahl der zu vereinigenden Ketten und dem Typ der jeweiligen passiven Elemente ergeben sich verschiedene Muster (Abb. 12).

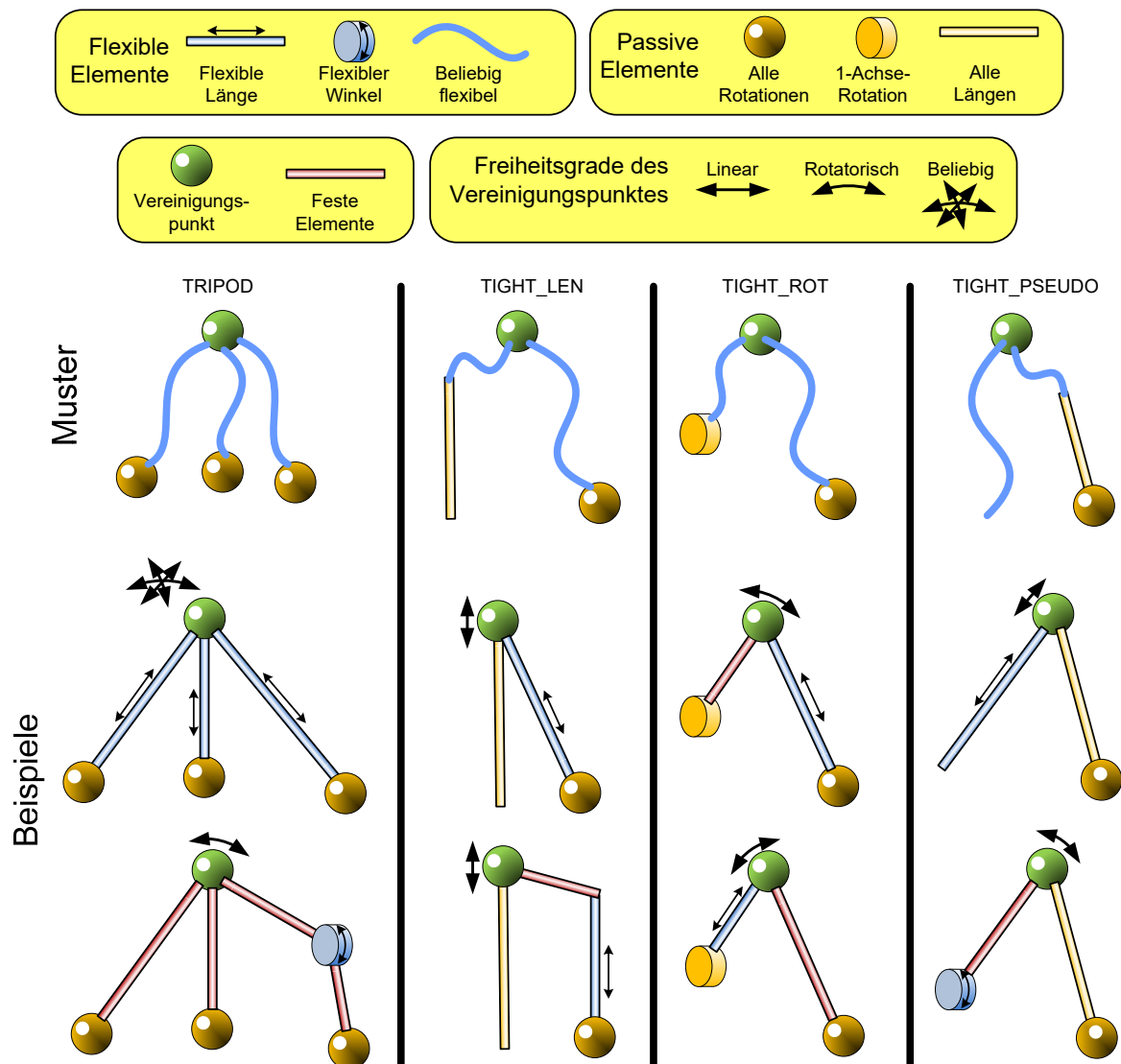


Abb. 12: Muster für geschlossene kinematische Ketten

Bisher werden folgende Muster unterstützt:

- **TRIPOD:** Drei Ketten werden in einem Punkt Vereinigungspunkt zusammengeführt. Jeder dieser Ketten muss beliebig um jeweils ein passives Element drehbar sein. Der Vereinigungspunkt wird berechnet über den Schnitt dreier Kugeloberflächen – die Kugelmittelpunkte sind die passiven Elemente und die Kugelradien ergeben sich über die jeweiligen Ketten.
- **TIGHT_LEN:** eine Kette enthält ein lineares passives Element. Daher kann sich der Vereinigungspunkt beliebig auf einer Geraden verschoben werden. Die zweite Kette muss um ein passives Element beliebig drehbar sein. Der Vereinigungspunkt wird berechnet über den Schnitt eine Kugeloberfläche und einer Gerade.
- **TIGHT_ROT:** eine Kette enthält ein drehbares passives Element, das nur die Drehung um eine Achse zulässt. Der befindet sich der Vereinigungspunkte auf dem Kreis um die Drehachse. Der Vereinigungspunkt wird berechnet über den Schnitt eine Kugeloberfläche und eines Kreises (3D).
- **TIGHT_PSEUDO:** Diese kinematische Kette ist nicht wirklich eine geschlossene Kette im ursprünglichen Sinn (daher der Zusatz *pseudo*). Gemeint ist damit: der Vereinigungspunkte wird komplett durch eine der Ketten definiert, da diese kein passives Element enthält. Die zweite Kette hängt "lose" am Vereinigungspunkt. Man kann die zweite Kette wie eine Art Sensor ansehen, der den Endpunkt der ersten Kette einmessen kann.

Weitere Muster sind denkbar, z.B. analog zu **TIGHT_ROT** ein Muster mit zwei 1-Achse-Rotationen aufbauen. Hierzu gibt es aber in der Regel überhaupt keine Position für den Vereinigungspunkt – diesen gibt es nur, wenn 1. die 1-Achse-Rotationen parallele Drehebene haben und 2. der Abstand der Ebenen so passt, dass sich die Teilketten treffen können. Das wird aber im Rahmen der Ungenauigkeiten einer Simulation nie eintreffen. Daher wurden nur Muster realisiert, die in der Regel eine Lösung haben.

5 Inverse Kinematik

5.1 Einleitung

Die inverse Kinematik dient einem Roboter, konkrete Bewegungen zu berechnen um eine bestimmte Zielposition des Greifers zu erreichen. Während man leicht die Endposition aus den Gelenkeinstellungen (Servo-Winkeln und Linear-Motor-Längen) berechnen kann (*direkte Kinematik*), ist die Umkehrung schwieriger.

Das Problem der *inversen Kinematik*: man hat nur für bestimmte Fälle eine geschlossene Lösung (siehe Abschnitt 5.2). Im allgemeinen Fall ist eine Näherungslösung erforderlich. Diese verwendet die direkte Kinematik (hierfür gibt es eine geschlossene Lösung), um eine bestimmte Einstellung zu "testen". D.h. es wird ermittelt, welches Resultat eine (zuerst beliebige) Einstellung hat, um sie mit dem Wunschresultat zu vergleichen. Danach wird die Einstellung so modifiziert, dass sie dem Wunsch etwas näher kommt. Nach einigen Iterationsschritten liegt (hoffentlich) eine Einstellung vor, die genau zu dem gewünschten Resultat führt.

Die Realisierung in dieser Umgebung basiert auf dem *Downhill-Simplex-Verfahren* (auch *Nelder-Mead-Verfahren* genannt). Vereinfacht ausgedrückt: es wird eine bestimmte Konstellation von Gelenkeinstellungen virtuell eingestellt, sowie einige Variationen um diese Einstellung herum. Zu jeder wird die Endposition über direkte Kinematik berechnet. Dann wird der Abstand zur gewünschten Position berechnet und versucht, den Fehler zu minimieren.

Eine andere Sicht: der Abstandsfehler ist eine Funktion über alle Gelenkeinstellungen. Für diese Funktion muss das (lokale) Minimum gesucht werden. Die Aufgabe kann also auf ein numerisches Verfahren zur Minimumssuche abgebildet werden. Als Startwert werden die alten Servo-Winkel genommen. Damit ist sichergestellt, dass das Minimum mit den geringsten Änderungen der Einstellungen erreicht wird.

5.2 Geschlossene Lösung am Beispiel 'Simple Arm'

Für den Spezialfall eines einfachen Arms (3DOF) gibt es eine geschlossene Lösung.

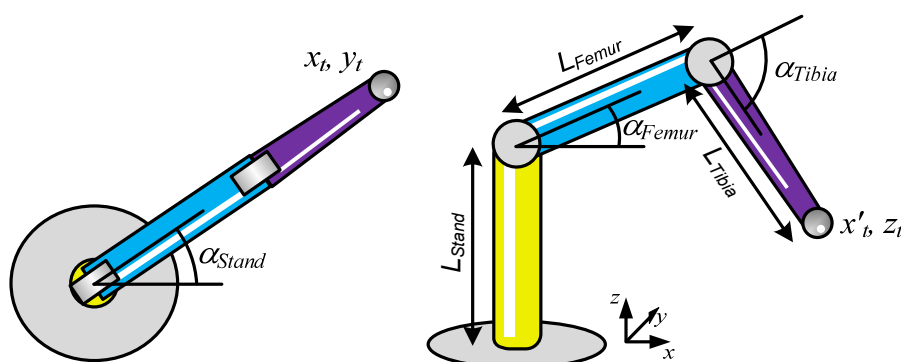


Abb. 13: Berechnungen am einfachen Arm

Neben dem senkrechten Segment ("Stand") gibt es zwei Segmente, die hier in Anlehnung an Insekten Beine *Femur* ("Oberschenkel") und *Tibia* ("Schienbein") genannt werden (Abb. 13).

Wir nehmen an, dass der Endpunkt x_t, y_t, z_t lokal in Relation zum Stand-Servo ausgedrückt wird. Weiter sind die Längen $L_{Stand}, L_{Femur}, L_{Tibia}$ bekannt. Gesucht werden die Servo-Winkel $\alpha_{Stand}, \alpha_{Femur}, \alpha_{Tibia}$.

Der Stand-Winkel kann leicht über die Umkehrung des Tangens ermittelt werden:

$$\alpha_{Stand} = \text{atan2}(y_t, x_t)$$

Man vereinfacht das resultierende Problem, indem man die Stand-Drehung zurücknimmt, d.h. den Arm in Richtung x -Achse dreht. Darüber hinaus legt man den Nullpunkt in den Femur-Servo (Abb. 14 links, d.h. die z -Koordinate wird um die Stand-Länge verringert. Damit

$$x'_t = \sqrt{x_t^2 + y_t^2}$$

$$z'_t = z - L_{Stand}$$

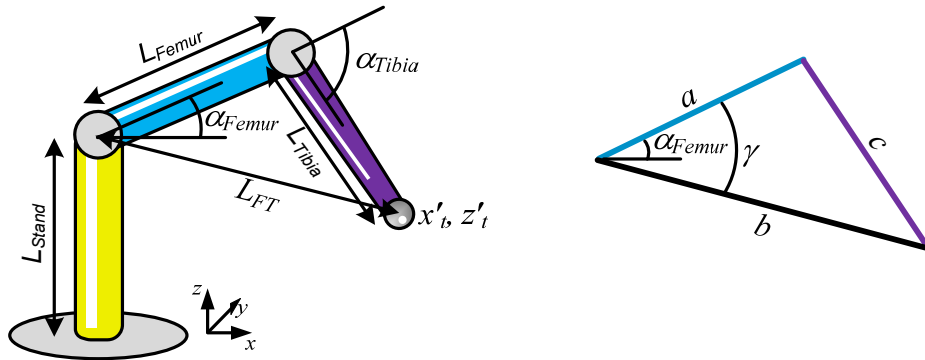


Abb. 14: Berechnung des Femur-Winkels

Darüber hinaus interessiert uns die Entfernung von Femur-Servo bis zum Zielpunkt L_{FT} . Es gilt

$$L_{FT} = \sqrt{x_t'^2 + z_t'^2}$$

Mit diesen Größen kann man den Cosinus-Satz anwenden (Abb. 14 rechts). Allgemein gilt für ein Dreieck

$$c^2 = a^2 + b^2 - 2ab \cos(\gamma)$$

Angewendet auf unsere Größen bedeutet das

$$L_{Tibia}^2 = L_{Femur}^2 + L_{FT}^2 - 2 \cdot L_{Femur} \cdot L_{FT} \cdot \cos(\gamma)$$

und damit

$$\gamma = \arccos\left(\frac{L_{Femur}^2 + L_{FT}^2 - L_{Tibia}^2}{2 \cdot L_{Femur} \cdot L_{FT}}\right)$$

Allerdings wollen wir ja nicht den aufspannenden Winkel γ berechnen, sondern nur den Winkel "über dem Horizont" α_{Femur} . Dieser ergibt sich, indem man den Winkel "unter dem Horizont" zum Touchpoint abzieht. Das führt zu

$$\alpha_{Femur} = \arccos\left(\frac{L_{Femur}^2 + L_{FT}^2 - L_{Tibia}^2}{2 \cdot L_{Femur} \cdot L_{FT}}\right) - \text{atan2}(z'_t, x'_t)$$

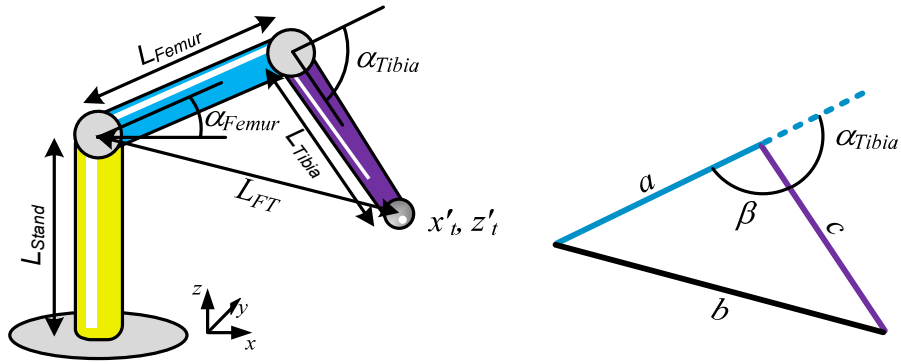


Abb. 15: Berechnung des Tibia-Winkels

Auch der Tibia-Winkel wird über den Cosinus-Satz berechnet. Hier gilt

$$b^2 = a^2 + c^2 - 2ac \cos(\beta)$$

und damit

$$L_{FT}^2 = L_{Femur}^2 + L_{Tibia}^2 - 2 \cdot L_{Femur} \cdot L_{Tibia} \cdot \cos(\beta)$$

und schließlich

$$\beta = \arccos\left(\frac{L_{Femur}^2 + L_{Tibia}^2 - L_{FT}^2}{2 \cdot L_{Femur} \cdot L_{Tibia}}\right)$$

Wir benötigen allerdings nicht den Innenwinkel sondern die Ergänzung zu 180° . Soll der Winkel wieder so ausgedrückt werden, dass höher bedeutet "über dem Horizont", erhält man

$$\alpha_{Tibia} = \arccos\left(\frac{L_{Femur}^2 + L_{Tibia}^2 - L_{FT}^2}{2 \cdot L_{Femur} \cdot L_{Tibia}}\right) - 180^\circ$$

5.3 Inverse Kinematik mit der Jacobi-Matrix

Die Jacobi-Matrix vermittelt zwischen den Gelenkzuständen (Servo oder Linearmotor) und der Endpose. In der allgemeinen Form wird die Pose als 3D-Position und 3D-Rotation ausgedrückt. Im ersten Ansatz wurde in der Realisierung allerdings nur die Position ohne Rotation gewertet. Da der Greifer über drei Servos mit dem Arm verbunden ist, kann man darüber jede Greiferpose einstellen.

Wir gehen also von einer Position p

$$p = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

aus. Die Gelenkzustände seien

$$q = \begin{pmatrix} q_1 \\ \dots \\ q_n \end{pmatrix}$$

Dann ist die Jacobi-Matrix

$$J(q) = \begin{pmatrix} \frac{\partial x(q)}{\partial q_1} & \frac{\partial x(q)}{\partial q_2} & \frac{\partial x(q)}{\partial q_3} & \dots & \frac{\partial x(q)}{\partial q_n} \\ \frac{\partial y(q)}{\partial q_1} & \frac{\partial y(q)}{\partial q_2} & \frac{\partial y(q)}{\partial q_3} & \dots & \frac{\partial y(q)}{\partial q_n} \\ \frac{\partial z(q)}{\partial q_1} & \frac{\partial z(q)}{\partial q_2} & \frac{\partial z(q)}{\partial q_3} & \dots & \frac{\partial z(q)}{\partial q_n} \end{pmatrix}$$

Es gilt dann

$$\dot{p} = J(q)\dot{q}$$

Da \dot{p} die zeitliche Ableitung der Position ist, können wir es auch mit v (also dem Symbol der Geschwindigkeit) bezeichnen.

Bevor wir den Nutzen von J darstellen, soll erst einmal geklärt werden, wie man sie erhält. Man könnte es analytisch berechnen – dazu müsste man aber für jeden Aufbau eines Arm-Bots die J von Hand erneut berechnen. Daher wurde der numerische Weg gewählt: anhand des internen Simulationsmodells kann die direkte Kinematik in geschlossener Form so-wieso schon automatisch beschafft werden. Sei

$$p = d(q)$$

die Funktion, die zu einem Zustand die Endposition berechnet. Dann kann man für ein kleines Inkrement h und einen Gelenkwinkel i folgendes berechnen:

$$\Delta p_i(q) = \frac{1}{2h} \left(d \begin{pmatrix} q_1 \\ q_i + h \\ q_n \end{pmatrix} - d \begin{pmatrix} q_1 \\ q_i - h \\ q_n \end{pmatrix} \right)$$

Für kleine h gilt

$$\Delta p_i(q) \approx \frac{\partial p(q)}{\partial q_i}$$

Damit kann man $J(q)$ wie folgt numerisch berechnen

$$J(q) = \begin{pmatrix} \Delta p_{1x}(q) & \Delta p_{2x}(q) & \Delta p_{3x}(q) & \dots & \Delta p_{nx}(q) \\ \Delta p_{1y}(q) & \Delta p_{2y}(q) & \Delta p_{3y}(q) & \dots & \Delta p_{ny}(q) \\ \Delta p_{1z}(q) & \Delta p_{2z}(q) & \Delta p_{3z}(q) & \dots & \Delta p_{nz}(q) \end{pmatrix}$$

Diese Berechnung ist über `DirectKinematics.getPositionJacobian(...)` aufrufbar. Jetzt zur inversen Kinematik. Aus

$$v = J(q)\dot{q}$$

folgt

$$\dot{q} = J^{-1}(q)v$$

wobei im Fall eines redundanten Manipulators die rechts-pseudo-Inverse Matrix von J verwendet wird. Unglücklicherweise ist man aber nicht an den Gelenkgeschwindigkeiten \dot{q} sondern an den Gelenkzuständen q interessiert. Man kann allerdings über \dot{q} integrieren, also

$$q(T) = q(0) + \int_0^T \dot{q}(t) dt = q(0) + \int_0^T J^{-1}(q(t))v(t) dt$$

Die Integration wird auch wieder numerisch gemacht und zwar wie folgt:

Seien

- q_0 : der aktuelle Gelenkzustand
- p : die Zielposition
- α : die Schrittweite

Dann lautet der Pseudo-Code wie folgt:

```

q ← q0
loop {
     $\tilde{p} \leftarrow d(q)$ 
     $\Delta p \leftarrow p - \tilde{p}$ 
    if  $\Delta p$  is small enough: finish
    scale  $\Delta p$  to  $\alpha$ 
     $\Delta q = J^{-1}(q)\Delta p$ 
     $q \leftarrow q + \Delta q$ 
}

```

6 Kommunikation und Ereignisse, Trigger

Im Gegensatz zu autonomen mobilen Robotern stehen bei Manipulatoren nicht die Sensoren zur Umwelterkennung im Vordergrund. Häufig werden Objekte gegriffen, ohne dass sie sensorisch erfasst wurden, einfach weil der Roboter 'weiß', dass ein entsprechendes Objekt an einer bestimmten Stelle liegen muss. Häufig kommen Industrie-Roboter ohne Erfassung der Umwelt aus, da die Umwelt so eingerichtet wurde, dass alle Bedingungen für die Roboter-Tätigkeit (Material, Werkzeuge) vorliegen. Sensoren melden eventuell nur Fehlersituationen.

Manchmal soll der Roboter aber doch auf die Umgebung reagieren. Ein Robot-Controller möchte beispielsweise über bestimmte Ereignisse in der Umgebung informiert werden (z.B. Ball liegt an einer bestimmten Stelle). Darüber hinaus möchten Robot-Controller einem anderen Robot-Controller Ereignisse melden (z.B. ich habe den Ball an einer bestimmten Stelle abgelegt). Letztlich gibt es Objekte der Simulation, die über Ereignisse gesteuert werden, z.B. können Förderbänder an- und ausgeschaltet werden.

Ein einfacher Kommunikationsmechanismus bildet beide Anwendungsfälle ab: der *Trigger*. Trigger sind mit Interrupt-Eingängen oder GPIO-Signalen an Boards (z.B. Arduino oder Raspberry PI) vergleichbar. Man kann sich vorstellen, dass der Erzeuger eines Ereignisses ein binäres Signal am Eingang auslöst und die Software führt dann Behandlungs-Code aus. Die unterschiedlichen Signale werden aber nur durch einen String unterschieden. Im Gegensatz zu echten Controller-Boards, sind alle **String**-Werte erlaubt. Es gibt zwei Arten, wie Trigger ausgelöst werden können:

- Im Robot-Controller per Code: hierzu ruft der Auslöser **sendTrigger(instanceNumber, triggerString)** auf.
- Per Umgebungsdatei: hierzu gibt es die Direktive **TRIGGER** (Seite 16). Die Simulationsumgebung sorgt dann automatisch dafür, dass das entsprechende Ereignis gemeldet wird.

Die Instanznummer beschreibt bei mehreren Robotern, welcher gemeint ist (0...). Sie entspricht also beispielsweise **<nr>** bei der **ROBOT<nr>**-Direktive. Die Instanznummer kann auch **-1** sein, dann bekommen alle Robot-Controller das Ereignis. Darüber hinaus gibt es Objekte (z.B. Förderbänder), die per Umgebungsdatei eine Instanznummer zugewiesen bekommen.

Im Fall eines Empfangs wird bei dem Robot-Controller die Methode **receiverTrigger(triggerString)** aufgerufen. Es wird sichergestellt, dass bei mehreren schnell aufeinanderfolgenden Triggern kein Ereignis vergessen wird, allerdings gibt es keine Gewähr für die Zustellungszeit. Es ist auch nicht möglich, an den Sender den Erfolg über die Behandlung zu melden, selbst Exceptions in der Behandlung werden nicht an zurückgemeldet.

Dieses einfache System ist für einige Anwendungen noch sehr unbequem. Beispielsweise soll ein Förderband abhängig von verschiedenen Bedingungen an- oder ausgeschaltet werden. Die Berechnung dieser Bedingungen könnte verschiedene Quellen berücksichtigen oder gar interne Status-Variablen verwalten. Darüber hinaus könnten Ereignisse an mehrere Instanzen verschickt werden. Die Umgebung bietet für die Formulierung verschiedener Bedingungen den so genannten *Trigger Processor* an. Dieser registriert sich als Empfänger für Trigger und produziert selbst neue Trigger auf der Basis von Programmcode. Dieser kann beliebig interne Variablen verwalten und beliebig komplexe Bedingungen ausdrücken.

7 Details zu Verfahren und Berechnungen

7.1 Anwendung einer Roto-Translation

Eine Roto-Translation besteht aus drei Drehungen und einer 3D-Translation (Abb. 16). Von den drei Rotationen werden jeweils die so genannten *Euler-Winkel* angegeben.

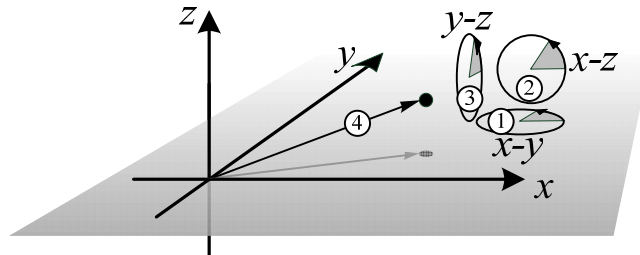


Abb. 16: Anwendung der Teile einer Roto-Translation

Die Bestandteile der Roto-Translation werden (insbesondere in der Simulation) in folgender Reihenfolge angewendet:

- Drehung um den Nullpunkt in der x - y -Ebene (also um die z -Achse),
- Drehung um den Nullpunkt in der x - z -Ebene (also um die y -Achse),
- Drehung um den Nullpunkt in der y - z -Ebene (also um die x -Achse),
- Translation

Wird die Drehung durch eine 3×3 -Rotationsmatrix ausgedrückt, so kann man sie durch das Produkt von drei einzelnen Rotationsmatrizen berechnen. Seien die Winkel a , b , c wie folgt definiert:

- a : Winkel der ersten Rotation in der x - y -Ebene um die z -Achse,
- b : Winkel der zweiten Rotation in der x - z -Ebene um die y -Achse,
- c : Winkel der dritten Rotation in der y - z -Ebene um die x -Achse.

Dann erhalten wir die drei Rotationsmatrizen

$$\begin{pmatrix} ca & -sa & \\ sa & ca & \\ & & 1 \end{pmatrix}, \begin{pmatrix} cb & -sb & \\ & 1 & \\ sb & & cb \end{pmatrix}, \begin{pmatrix} 1 & & \\ & cc & -sc \\ & sc & cc \end{pmatrix}$$

Hierbei sollen sa , sb , sc jeweils $\sin(a)$, $\sin(b)$, $\sin(c)$ bedeuten, ca , cb , cc analog $\cos(a)$, $\cos(b)$, $\cos(c)$.

Beim Produkt wird die Matrix "rechts" als erstes angewendet. Damit lautet die Gesamt-Rotationsmatrix

$$\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} = \begin{pmatrix} 1 & & \\ & cc & -sc \\ & sc & cc \end{pmatrix} \cdot \begin{pmatrix} cb & -sb & \\ & 1 & \\ sb & & cb \end{pmatrix} \cdot \begin{pmatrix} ca & -sa & \\ sa & ca & \\ & & 1 \end{pmatrix}$$

$$\begin{aligned}
&= \begin{pmatrix} 1 & & \\ & cc & -sc \\ & sc & cc \end{pmatrix} \cdot \begin{pmatrix} ca \cdot cb & -sa \cdot cb & -sb \\ sa & ca & \\ ca \cdot sb & -sa \cdot sb & cb \end{pmatrix} \\
&= \begin{pmatrix} ca \cdot cb & -sa \cdot cb & -sb \\ sa \cdot cc - ca \cdot sb \cdot sc & ca \cdot cc + sa \cdot sb \cdot sc & -cb \cdot sc \\ sa \cdot sc + ca \cdot sb \cdot cc & ca \cdot sc - sa \cdot sb \cdot cc & cb \cdot cc \end{pmatrix}
\end{aligned}$$

Häufig möchte man aus einer einmal berechneten Rotationsmatrix die einzelnen Euler-Winkel zurückrechnen. Das geht über die Zusammenhänge

$$\arctan 2(\alpha) = \frac{\sin(\alpha)}{\cos(\alpha)} \text{ und } \sin^2(\alpha) + \cos^2(\alpha) = 1$$

Daraus erhält man

$$\begin{aligned}
c &= -\arctan 2\left(\frac{r_{23}}{r_{33}}\right) = -\arctan 2\left(\frac{-cb \cdot sc}{cb \cdot cc}\right) = \arctan 2\left(\frac{sc}{cc}\right) \\
b &= \arctan 2\left(\frac{-r_{13}}{\sqrt{r_{23}^2 + r_{33}^2}}\right) = \arctan 2\left(\frac{sb}{\sqrt{cb^2 \cdot sc^2 + cb^2 \cdot cc^2}}\right) = \arctan 2\left(\frac{sb}{\sqrt{cb^2 \cdot (sc^2 + cc^2)}}\right) \\
&= \arctan 2\left(\frac{sb}{cb \sqrt{sc^2 + cc^2}}\right) = \arctan 2\left(\frac{sb}{cb \cdot 1}\right) = \arctan 2\left(\frac{sb}{cb}\right) \\
a &= -\arctan 2\left(\frac{r_{12}}{r_{11}}\right) = -\arctan 2\left(\frac{-sa \cdot cb}{ca \cdot cb}\right) = \arctan 2\left(\frac{sa}{ca}\right)
\end{aligned}$$

Diese Berechnungen sind in der Methode **Matrix.eulerAngles()** hinterlegt.

Als letztes: Die Reihenfolge der Anwendung der einzelnen Drehungen ist wichtig bei der Berechnung der Gesamt-Rotationsmatrix. Man könnte statt in der Reihenfolge oben (Drehung x - y -Ebene – x - z -Ebene – y - z -Ebene) dieselbe Rotation über Drehungen der umgekehrten Reihenfolge (Drehung y - z -Ebene – x - z -Ebene – x - y -Ebene) erhalten. Man kann dies analog zu oben aus einer einmal berechneten Rotationsmatrix zurückrechnen, und zwar über

$$\begin{aligned}
\begin{pmatrix} r_{11} & r_{12} & r_{13} \\ r_{21} & r_{22} & r_{23} \\ r_{31} & r_{32} & r_{33} \end{pmatrix} &= \begin{pmatrix} ca & -sa & \\ sa & ca & \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} cb & & -sb \\ & 1 & \\ sb & & cb \end{pmatrix} \cdot \begin{pmatrix} 1 & & \\ & cc & -sc \\ & sc & cc \end{pmatrix} \\
&= \begin{pmatrix} ca & -sa & \\ sa & ca & \\ & & 1 \end{pmatrix} \cdot \begin{pmatrix} cb & -sb \cdot sc & -sb \cdot cc \\ & cc & -sc \\ sb & cb \cdot sc & cb \cdot cc \end{pmatrix} \\
&= \begin{pmatrix} ca \cdot cb & -ca \cdot sb \cdot sc - sa \cdot cc & -ca \cdot sb \cdot cc + sa \cdot sc \\ sa \cdot cb & -ca \cdot sb \cdot sc + ca \cdot cc & -sa \cdot sb \cdot cc - ca \cdot sc \\ sb & cb \cdot sc & cb \cdot cc \end{pmatrix}
\end{aligned}$$

$$c = \arctan 2 \left(\frac{r_{32}}{r_{33}} \right) = -\arctan 2 \left(\frac{cb \cdot sc}{cb \cdot cc} \right) = \arctan 2 \left(\frac{sc}{cc} \right)$$

$$b = \arctan 2 \left(\frac{r_{31}}{\sqrt{r_{32}^2 + r_{33}^2}} \right) = \arctan 2 \left(\frac{sb}{\sqrt{cb^2 \cdot sc^2 + cb^2 \cdot cc^2}} \right) = \arctan 2 \left(\frac{sb}{\sqrt{cb^2 \cdot (sc^2 + cc^2)}} \right)$$

$$= \arctan 2 \left(\frac{sb}{cb \sqrt{sc^2 + cc^2}} \right) = \arctan 2 \left(\frac{sb}{cb \cdot 1} \right) = \arctan 2 \left(\frac{sb}{cb} \right)$$

$$a = \arctan 2 \left(\frac{r_{21}}{r_{11}} \right) = \arctan 2 \left(\frac{sa \cdot cb}{ca \cdot cb} \right) = \arctan 2 \left(\frac{sa}{ca} \right)$$

Diese Berechnungen sind in der Methode **Matrix.eulerAnglesReverse()** hinterlegt.

7.2 Berechnung einer Rotation zwischen zwei Punkten

Bei der Berechnung von geschlossenen kinematischen Ketten ergibt sich ein Problem: gegeben zwei Punkte, die durch eine 3D-Rotation um den Nullpunkt aufeinander abgebildet werden können, d.h. sie haben dieselbe Entfernung zum Nullpunkt. Wir suchen eine 3D-Rotation, die das bewerkstelligt.

Unglücklicherweise gibt es unendlich viele Rotationen zwischen zwei Punkten, so dass wir weitere Bedingungen brauchen. Hier wird daher gefordert, dass die Rotation um die x-Achse (Nummer 3 in Abb. 16 auf Seite 36), also der Rollwinkel gleich 0 ist. Darüber hinaus sollen die anderen beiden Rotationen in der Reihenfolge

- erst in der x-y-Ebene,
- dann in der x-z-Ebene.

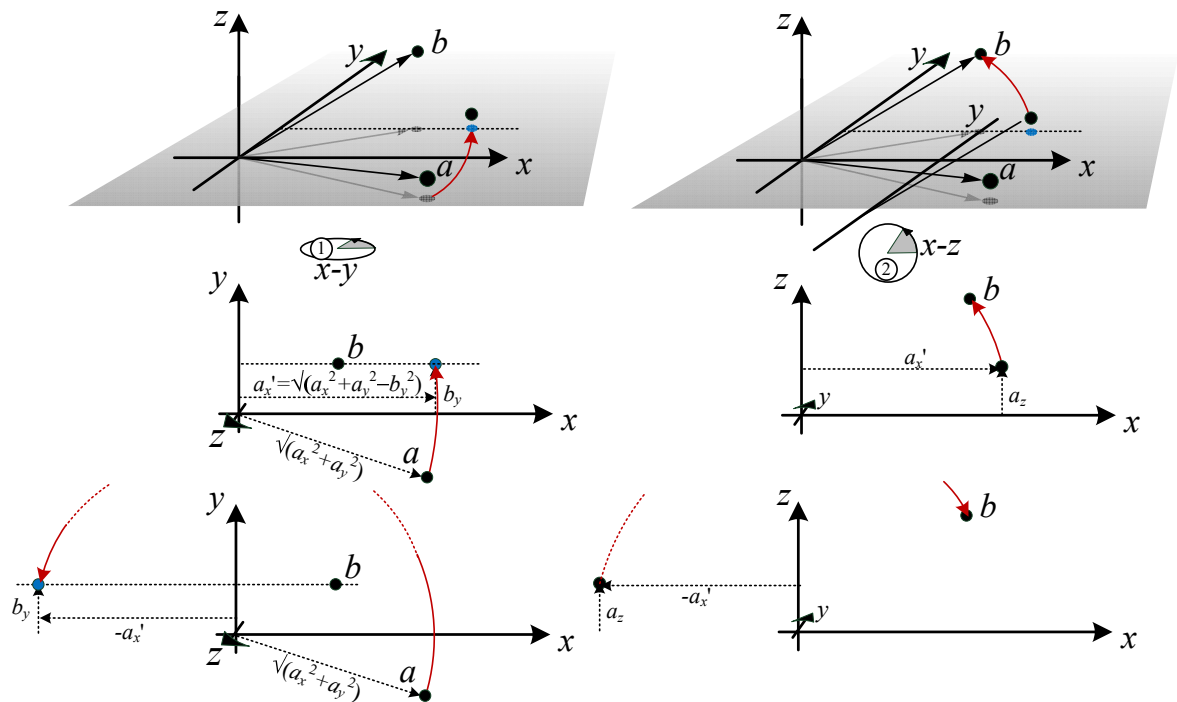


Abb. 17: Rotation zwischen zwei Punkten

Abb. 17 illustriert das Vorgehen. Der Originalpunkt sei a und der Zielpunkt nach der Rotation von a sei b . Wir führen erst die Gier-Rotation in der x - y -Ebene aus. Ziel der Projektion von a in die x - y -Ebene nach der ersten Gier-Rotation ist, den y -Wert von b anzunehmen. Nur dann kann durch eine weitere Nick-Rotation (bei der der y -Wert sich nicht verändert) b erreicht werden. Nach der Gier-Rotation sei die Position von a also (a_x', b_y, a_z) , wobei

$$a_x' = \sqrt{a_x^2 + a_y^2 - b_y^2}$$

Leider ist aber auch $-a_x'$ eine Lösung für die Gier-Rotation von a (Abb. 17 links unten). So dass wir die beiden Zielpunkte (a_x', b_y, a_z) und $(-a_x', b_y, a_z)$ bekommen. Die beiden fraglichen Gier-Winkel zur Rotation bekommt man

$$\alpha_{xy1} = \arctan 2(b_y, a_x') - \arctan 2(a_y, a_x), \quad \alpha_{xy2} = \arctan 2(b_y, -a_x') - \arctan 2(a_y, a_x)$$

Der Nickwinkel wird letztlich über

$$\alpha_{xz1} = \arctan 2(b_z, b_x) - \arctan 2(a_z, a_x'), \quad \alpha_{xz2} = \arctan 2(b_z, b_x) - \arctan 2(a_z, -a_x')$$

berechnet. Diese Vorgehensweise ist in der Methode **Geom.angXYangXZtoMapPositions(...)** realisiert.

Darüber hinaus kann derselbe Ansatz gewählt werden, um eine entsprechende 3D-Rotation zu berechnen mit

- erst in der x - z -Ebene,
- dann in der x - y -Ebene.

(weiterhin Rollwinkel gleich 0). Die Berechnungen oben sind identisch, nur dass y und z vertauscht werden. Die resultierende Vorgehensweise ist in der Methode **Geom.angXYangXZtoMapPositionsReverse(...)** realisiert.

8 Weitere technische Details

8.1 Java-Module

Ab Java 9 wurde das Einbinden von Packages überarbeitet und durch ein Modulkonzept erweitert. Da auf dem realen Roboter weiterhin Java 8 verwendet wird, ist ein zweigleisiger Ansatz notwendig, der

- einerseits Module verwendet, wenn es unterstützt wird (insb. im Simulator),
- andererseits die Module ignoriert, wenn es nicht unterstützt wird,
- dieselbe Quellenbasis verwendet.

Glücklicherweise sind die Varianten formal nicht sehr unterschiedlich. Die Modul-Variante verlangt

- ein Verzeichnis für alle Packages, die in diesem Modul liegen – nur-Packages (ohne Module) liegen demgegenüber in '.' und das Package-Verzeichnis ist direkt die erste Ebene des Packages,
- eine Datei `module-info.java`, die die Meta-Eigenschaften eines Moduls beschreibt.

Damit kann man eine nicht-Modul-Variante verwenden, indem man `module-info.java` ignoriert und den `CLASSPATH` in jedes Modulverzeichnis legt (also auf den Start der Packages).

Die Simulationsumgebung wurde komplett auf Module umgeschrieben. Hierbei ist relevant, dass Module keine zirkulären Abhängigkeiten aufweisen dürfen. Deshalb waren einige Umarbeitungen notwendig. Als Resultat wurden folgende Module eingerichtet:

- **Platform**: Funktionen, die zwischen Simulation und echtem Roboter unterschiedlich sind, z.B. Logging und Uhrzeit,
- **Basics**: grundlegende Berechnungen (**math**), Punktwolken (**points**) etc.,
- **Robotinterface**: das Package `robotinterface`; eine abstrakte Schnittstelle auf alle Roboter-Einrichtungen,
- **Simulator**: package `kin` (Simulator),
- **Manipulator**: virtuelle Modell der Manipulatoren,
- **Robotcontroller**: alle Robot-Controller.

Abb. 18 zeigt den Abhängigkeitsgraph dieser Module.

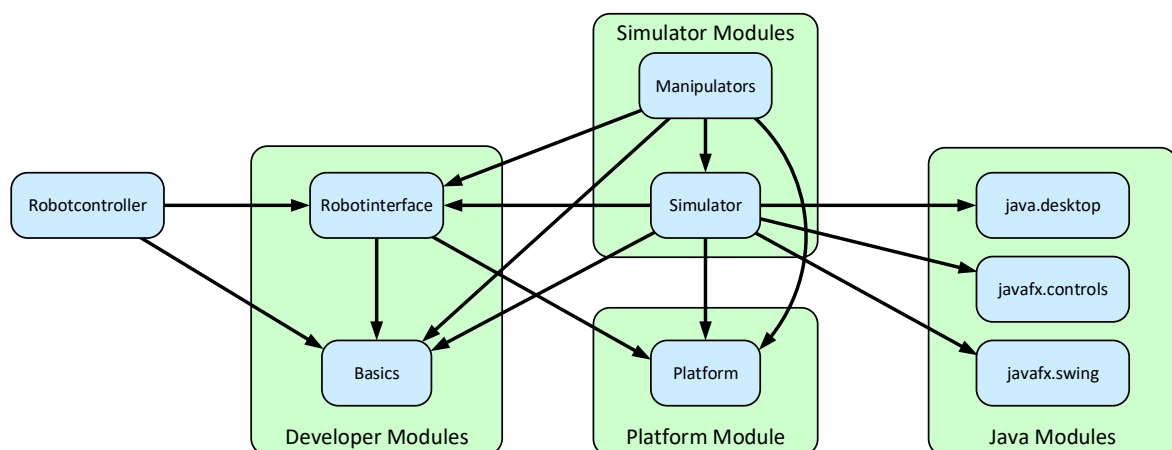


Abb. 18: Modulgraph der Simulationsumgebung

Entgegen der Java-Gepflogenheiten wurden die Modulnamen groß geschrieben, um sie von den gleichlautenden Packages zu unterscheiden.

Die Zerlegung in Module soll ausschließlich auf die Umgebung selbst angewendet werden, *nicht* auf die Entwicklung von Robot-Controllern. Jeder Robot-Controller sollte sich komplett in dem Modul **Robotcontroller** befinden, das ausschließlich die Module **Basics** und **Robotinterface** verwenden darf.

8.2 JavaFX

Mit Java 11 wurde JavaFX aus dem Standard-Java entfernt. JavaFX (3D) wird im Simulator benötigt, um die 3D-Ansicht im Hauptfenster darzustellen. Die JavaFX-Bibliotheken müssen dann separat beschafft werden (<https://openjfx.io/>). Die Einbindung erfolgt über das in Java 9 eingeführte Modulsystem. Sowohl der Build-Prozess als auch beim Start des Simulators müssen die entsprechenden Module über den Zusatz

```
--module-path C:\javafx-sdk-11.0.2\lib  
--add-modules javafx.controls,javafx.swing
```

referenziert werden, wobei der Pfad durch Installationsverzeichnis der JavaFX-Bibliotheken ersetzt werden muss.

Man kann allerdings die Verwendung von JavaFX 3D komplett umgehen. Über die Start-Option **-render native** wird ein internes Verfahren für die 3D-Ansicht verwendet, das nur natives Java benötigt.

Wird das Generieren der Software gemäß des Java-Modulkonzeptes durchgeführt, so werden die Java-FX-Module schon eingebunden.

9 Offene Probleme, noch nicht realisierte Funktionen

Dies ist eine lockere Sammlung von "to-dos" für die Zukunft.

- Die Simulation der rollenden und reflektierenden Bälle ist nie richtig umgesetzt worden. In den meisten Fällen funktioniert das sehr realistisch, allerdings führt eine Vereinfachung häufig zu Problemen: pro Simulationsschritt wird einem Ball der Geschwindigkeitsvektor auf der Basis der Erdbeschleunigung hinzugefügt. Hat sich der Ball dann in ein Hindernis bewegt, wird es einfach wieder angehoben, bis er es nur noch berührt. Damit kann man sowohl den freien Fall, als auch die Schräge über einen Mechanismus abwickelt. Das Problem: liegt ein Ball auf einer Kante auf, so ergibt dieses Verfahren einen neuen Geschwindigkeitsimpuls von der Kante weg. Das wiederum führt dazu, dass dem Ball ständig "Energie" hinzugefügt wird. Das sieht man daran, dass der Ball beispielsweise in einer Senke liegt, dann aber plötzlich von einer Kante wegbeschleunigt wird. Genau genommen geht das Problem tiefer: liegt ein Ball auf einer Fläche, so wird dieser Ständig nach oben beschleunigt (da er gewissermaßen ständig von der Fläche reflektiert wird). Dieser Effekt ist allerdings kaum sichtbar, da diese Geschwindigkeit zu anfangs sehr klein ist, damit den Ball nur gering anhebt – korrekt ist es aber dennoch nicht.

Langfristig müsste die Simulation der Bälle überarbeitet werden. Allerdings sind die Bälle sowieso ein Sonderfall (die einzige voll rotationssymmetrische Form, daher einfacher zu simulieren). Daher wäre sowieso die Simulation anderer Formen, die der Greifer beeinflussen kann, wünschenswert.

Darüber hinaus ist der Ansatz in Abschnitt 4.2.1 (Seite 24) noch suboptimal: es werden zwei Fälle betrachtet, die sich aus Anzahl der Kollisionspunkte und Art der Kollisionspunkte (Touchpoint oder nicht) ergeben. Das basiert darauf, dass im Grunde genommen folgende Effekte über einen einzigen Ansatz simuliert werden sollten: freier Fall, Rollen auf schiefer Ebene und aufgezwungene Bewegung durch Greifer. Vielleicht müssen diese Effekte aber alle isoliert erkannt und simuliert werden.

- Die Simulation der Gravitation von Cuboiden ist noch nicht immer korrekt. Das Problem kann man auf `PolyhedricObstacle.moveOutObstacle(...)` runterbrechen. Hier ist das Problem, dass man keinen echten Überlappungstest zwischen Polyedern hat, sondern nur probiert, Eckpunkte als Kandidaten für Kollisionspunkte zu identifizieren. Bei vielen Überlappungen sind die Kollisionspunkte aber keines der Eckpunkte der beteiligten Polyeder. Die Vereinfachung, die der Methode zugrunde liegt, führt dadurch zu "falsch positiven" Kollisionen. Das zeigt sich beispielsweise daran, dass ein Fließband einen Quader über den Kippunkt hinaus schiebt, bevor er runterfällt.
- Das Hauptfenster wird ca. alle 10 Mal nicht geöffnet (Blockade im Aufruf `setVisible(true)`, der in `Frame.addNotify` stecken bleibt). Behebungsversuche waren bisher erfolglos. Ein neuer Start-Versuch über die Kommandozeile funktioniert meistens, während ein zweiter Versuch im Programm zur selben Blockade führt. Dieses Problem kann mehrere Ursachen haben, womöglich in der alten Java-Version 8. Das Problem ist auch erst entstanden, nachdem das Programm auf einem neuen Rechner mit anderem Grafiksystem übertragen wurde – vielleicht liegt es also an 3D-Eigenschaften der Grafikkarte. Auf neuen Installationen (Java 13) ist das Problem nicht mehr aufgetreten.
- Inverse Kinematik: im Moment stellt die Inverse Kinematik (z.B. `robotinterface.invkin.InverseKinematics`) nur eine Funktion bereit, die den Anknüpfungspunkt zum Greifer beschreibt. Es wird davon ausgegangen, dass an diesem Punkt der Greifer so montiert wurde, dass dieser sich in alle Richtungen drehen kann. Das ist aus zweier-

lei Hinsicht ungünstig. 1. Die Greifposition selbst wird nicht beschrieben, sondern die Position, an der der Greifer montiert ist (ohne dass man hier die Greifer-"Weite" kennt). 2. Die Pose an der Endposition kann nicht vorgegeben werden. Aus 2. ergeben sich Probleme, wenn a) der Greifer nicht beliebig am Anknüpfungspunkt gedreht werden kann, aber wichtiger b), wenn die 3 Raumrichtungen sich nicht durch eine universelle Rotationskomponente ergeben, sondern durch den Aufbau des Manipulators. Denkbar wäre ja, dass die insgesamt 6DOF sich durch eine komplizierte Konstruktion ergeben, bei der der Greifer vielleicht am Ende nur noch um die eigene Achse gedreht werden muss.

Kurz: die Inverse Kinematik müsste so erweitert werden, dass man die Greiferpose (6DOF) eingeben kann.

10 Abschlussarbeiten und IT-Projekte

bisher keine

11 Weitere Referenzen

bisher keine

12 Index

- 3DOF-Bein 30
- Body 21
- Commander 10
- Connectionpoint 21
- Cosinus-Satz 31
- Debug-Out 10
- direkte Kinematik 30
- Downhill-Simplex-Verfahren 30
- Elastischer Stoß 25
- Euler-Winkel 36
- Femur 30
- fix-side 21
- Fix-Side-Part 21
- Inverse Kinematik 30
 - mit Jacobi-Matrix 32
- Jacobi-Matrix 32
- Java
 - JavaFX 3D 20, 41
 - Module 40
- Kinematische Kette
 - geschlossene 26
 - offene 26
- Kommando-Programm 9
- Layer
 - Dynamic ~ 21
 - Model ~ 21
 - Render ~ 22
- Nelder-Mead-Verfahren 30
- rechts-pseudo-Inverse 34
- Robot-Controller 6, 7, 13, 14, 18, 19
 - Konfiguration 8, 13, 14, 19, 20
- Segment 21
- Skills 8
- tail-side 21
- Tail-Side-Part 21
- Tibia 30
- Touchpoint 21
- Trigger 16, 35
 - Console 10
 - Processor 35
- Umgebungsdatei 11, 18