View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Online Publikationen der Universität Stuttgart

Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 74

Motion Designer: ein interaktives Tool zum Entwerfen von Bewegungen

Philipp Niethammer

Studiengang: Softwaretechnik

Prüfer/in: Prof. Dr. Marc Toussaint

Betreuer/in: M.Sc. Stefan Otte

Beginn am: 12. Juni 2013

Beendet am: 12. Dezember 2013

CR-Nummer: I.2.9

Kurzfassung

Roboter werden zunehmend außerhalb abgetrennter und vollständig kontrollierter Umgebungen eingesetzt. Meist bedeutet dies, dass sie in der unmittelbaren Nähe von Menschen agieren. Dementsprechend muss bei der Entwicklung solcher Roboter dafür Sorge getragen werden, dass sie auf sich verändernde Situationen reagieren können, um mit Menschen zusammenzuarbeiten und diese nicht zu gefährden.

Neben einer ausgereiften Sensorik zur Erkennung und Beobachtung der Umgebung ist dafür insbesondere wichtig, eine Bewegung so auszuführen, dass das gewünschte Ziel erreicht und gleichzeitig allen etwaigen Hindernissen ausgewichen wird. Dafür muss die Bewegung in einer Form beschrieben werden, die dem Roboter erlaubt, sie der aktuellen Situation anzupassen. Um zum Beispiel Kollisionen mit Personen zu vermeiden, muss eine geeignete Bewegung des Weiteren schnell genug gefunden werden. Diese Problematik wird oft als Motion Generation beschrieben.

Um die Forschung in diesem Bereich zu unterstützen, sind Simulationen eines Szenarios ein wichtiges Kontrollmedium. Zusammen mit einer einfach zu bedienenden Oberfläche zum Erstellen einer Beschreibung können sie dabei helfen, gute Lösungen für ein Problem zu finden.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 Motivation . . . 7
- 1.2 Ziel der Arbeit . . . 8
- 1.3 Gliederung . . . 8

2 Grundlagen **11**
- 2.1 Begriffsklärung . . . 11
- 2.2 Generieren von Bewegungen . . . 12
  - 2.2.1 Problemstellung . . . 12
  - 2.2.2 Planung . . . 13

3 Stand der Technik **15**
- 3.1 Related Work . . . 15
  - 3.1.1 Algorithmen zum Planen der Bewegung . . . 15
  - 3.1.2 MoveIt! . . . 17
- 3.2 Gauss-Newton-basierter Optimierer . . . 18
  - 3.2.1 Kosten . . . 18
- 3.3 OpenRobotSimulator ORS . . . 21
  - 3.3.1 Der ORS-Graph . . . 21
  - 3.3.2 Motion . . . 22

4 Beschreibung von Bewegungsabläufen **25**
- 4.1 Motion Problem Project . . . 26
- 4.2 Task Beschreibung . . . 27
  - 4.2.1 TaskMap . . . 27
  - 4.2.2 TaskCosts . . . 28
- 4.3 TaskMapType Beschreibung . . . 29

5 MotionDesigner **33**
- 5.1 Motivation . . . 33
- 5.2 Zielgruppe . . . 34
- 5.3 Benutzeroberfläche . . . 34
  - 5.3.1 Allgemeiner Aufbau . . . 34
  - 5.3.2 Parametereditoren . . . 37
- 5.4 Implementierung . . . 40
  - 5.4.1 Model . . . 40
  - 5.4.2 View . . . 43

5.4.3 Trennung von Model und View 43

6 Zusammenfassung und Ausblick **47**
6.1 Ausblick 48

Literaturverzeichnis **49**

Abbildungsverzeichnis

2.1 Verschiedene Typen von Gelenken mit null bis sechs DoF (Quelle: [Tou13a]) . . 11
2.2 Ein primitiver Roboter mit zwei Freiheitsgraden und ein Hindernis und deren Repräsentation im ControlSpace (Quelle: [Hla10b, S. 40]) 14

3.1 Ein Beispiel für Cell Decomposition: Der Zustandsraum wird in Trapeze aufgeteilt und im daraus enstehenden Adjazenzgraph nach einem Weg vom Start zum Ziel gesucht (Quelle: [Hla10a, S. 40]). 16
3.2 MoveIt! beim Erstellen einer Bewegung mit Darstellung der Wahrnehmung des PR2s (Quelle: [SC13]) 17
3.3 Vereinfachtes Klassendiagramm des ORS-Moduls `Motion` 22

4.1 Entity-Relationship-Diagramm der gesamten Projektbeschreibung 27

5.1 MotionDesigner Hauptfenster mit einem klatschenden PR2 35
5.2 Einstellungen des Optimierers 35
5.3 Für die Zwischenstandsanzeige des Optimierers muss die „Gesprächigkeit“ erhöht werden. 35
5.4 Detailansicht eines Item-Parameters 38
5.5 Detailansicht für eine Liste von Shapes 39
5.6 Detailansicht für Tensoren 39
5.7 Im Model von MotionDesigner verwendete Umsetzung des Observer-Patterns 42

Verzeichnis der Listings

4.1 Beschreibung eines Tasks bestehend aus Metadaten, TaskMap und TaskCosts . 28
4.2 Beschreibung eines TaskMapTypes 29

1 Einleitung

1.1 Motivation

Der Einsatz von Robotern ist bereits heute aus dem Alltag nicht mehr wegzudenken. Sie kommen überall dort zum Einsatz, wo in großer Serie möglichst schnell ein Arbeitsschritt in gleichbleibend hoher Qualität getätigt werden muss, wie beispielsweise das Fertigen einer Schweißnaht im Automobilbau.

Klassische Industrieroboter, wie sie zur Zeit primär eingesetzt werden, agieren dabei aus Sicherheitsgründen in abgegrenzten Bereichen. Sie verfügen über keine Vorrichtungen zur Überwachung ihres Bewegungsraumes und arbeiten mit sehr hohen Kräften. Sollte sich unerwartet ein Objekt im Weg des Roboters befinden, kommt es zwangsläufig zu einer Kollision.

Die aktuelle Entwicklung zielt immer stärker in Richtung eines kooperativen Einsatzes von Mensch und Roboter. So plant der Autobauer BMW laut [Kni13] Roboter und Mitarbeiter parallel arbeiten zu lassen, wobei der Roboter sowohl den Mitarbeiter unterstützen als auch mühsame Arbeiten komplett übernehmen kann. Auch außerhalb industrieller Fertigungen finden sich zunehmend Beispiele. Nachdem im US-amerikanischen Bundesstaat Nevada bereits im Mai 2012 das erste autonome Testfahrzeug zugelassen wurde ([DMV12]) kündigt nun Volvo an, ab 2017 insgesamt 100 autonome Fahrzeuge im schwedischen Göteborg auf die Straßen bringen zu wollen ([Vol13]). Ein weiteres sehr neues Feld sind Assistenzroboter für Menschen mit Behinderung, die Unterstützung in alltäglichen Aufgaben wie dem Zubereiten von Mahlzeiten bieten, wie beispielsweise der an der Universität Bremen entwickelte FRIEND (Functional Robot arm with user-frIENdly interface for Disabled people, [IAT13]).

All diese Systeme haben eines gemeinsam: Sie müssen die Situation in ihrer Umgebung erkennen und rechtzeitig darauf reagieren.

Durch Isolation des Arbeitsbereichs kann ein Industrieroboter einem festen Bewegungsablauf folgen. Sensoren können sicherstellen, dass sich etwa das Werkstück an einer exakten Position befindet, sodass ein Roboter immer exakt die gleiche Bewegung ausführen kann. Diese Bewegung wird einmalig vor Inbetriebnahme durch Fachkräfte erstellt und dann ohne Abweichung vom Roboter ausgeführt. Dabei spielt die Zeit, die zum Erstellen der Bewegung erforderlich ist, für die Ausführung keine Rolle.

Interagiert der Roboter mit Objekten in seiner Umgebung ist solch ein deterministischer Bewegungsablauf nicht möglich. Stattdessen muss in Echtzeit ein Bewegungsablauf berechnet werden, der bestimmten Bedingungen entspricht.

Ein Roboter, der den Mitarbeiter eines Autobauers dabei unterstützt, ein schweres Bauteil exakt auszurichten, muss beispielsweise der Führung des Mitarbeiters folgen und vielleicht sogar die zukünftige Bewegung zu einem gewissen Grad antizipieren. Ein Assistenzroboter muss beim Anreichen von Speisen unter anderem die Position und Bewegung des Mundes verfolgen und autonome Fahrzeuge müssen unerwartet auftauchenden Hindernissen wie Fußgängern rechtzeitig ausweichen, um Unfälle zu verhindern.

Aus diesem Grund erfährt die Entwicklung von Systemen, die Bewegungen anhand von Bedingungen generieren, gesteigertes Interesse in der Industrie und Forschung. Dabei spielt neben der Güte der Bewegung insbesondere auch die für die Planung notwendige Zeit eine bedeutende Rolle.

Ein weiterer wesentlicher Punkt der Forschung ist die Beschreibung der Bewegung selbst. Diese muss hinreichend genau sein, um keine wichtigen Aspekte zu vergessen, aber gleichzeitig genug Spielraum lassen, um den Roboter flexibel reagieren zu lassen und auch die für die Planung nötige Zeit gering halten zu können.

1.2 Ziel der Arbeit

Diese Arbeit beschäftigt sich mit der Beschreibung solcher dynamisch definierten Bewegungen.

Das Ziel ist das Erstellen einer grafischen Oberfläche, die Entwickler darin unterstützt, eine gute Bewegung mit Hilfe von Simulationen zu erzeugen und durch geeignete Visualisierung der Ergebnisse zu bewerten. Eine geeignete Schnittstelle soll dabei den Entwickler bei der Beschreibung einzelner Kriterien entlasten und den nötigen Umfang des domänenspezifischen Wissens veringern.

Des Weiteren ist ein Datenformat zu finden, durch das die vollständige Beschreibung abgebildet wird. Dieses Datenformat soll durch Benutzer direkt lesbar sein und von Programmen genutzt werden können, um die Beschreibung und daraus resultierende Bewegung exakt zu rekonstruieren.

Grundlage dafür ist das in Kapitel 3.3 vorgestellte ORS-Framework, das Funktionalitäten zur Repräsentation von Robotern und ihrer Umgebung, der Simulation sowie dem programmatischen Beschreiben und Planen von Bewegungen enthält.

1.3 Gliederung

Die weitere Arbeit ist in folgender Weise aufgebaut:

Kapitel 2 – Grundlagen: Hier werden werden die Grundlagen dieser Arbeit beschrieben.

Kapitel 3 – Stand der Technik befasst sich mit aktuell vorhandenen Arbeiten (Related Work) und der ORS-Bibliothek

Kapitel 4 – Beschreibung von Bewegungsabläufen: Stellt das Datenformat zum Beschreiben eines Problems vor.

Kapitel 5 – MotionDesigner: Hier wird das entstandene Programm MotionDesigner vorgestellt.

Kapitel 6 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

2 Grundlagen

2.1 Begriffsklärung

In diesem Abschnitt sollen einige Begriffe im Zusammenhang der Robotik erläutert werden, die im Verlauf dieser Arbeit verwendet werden.

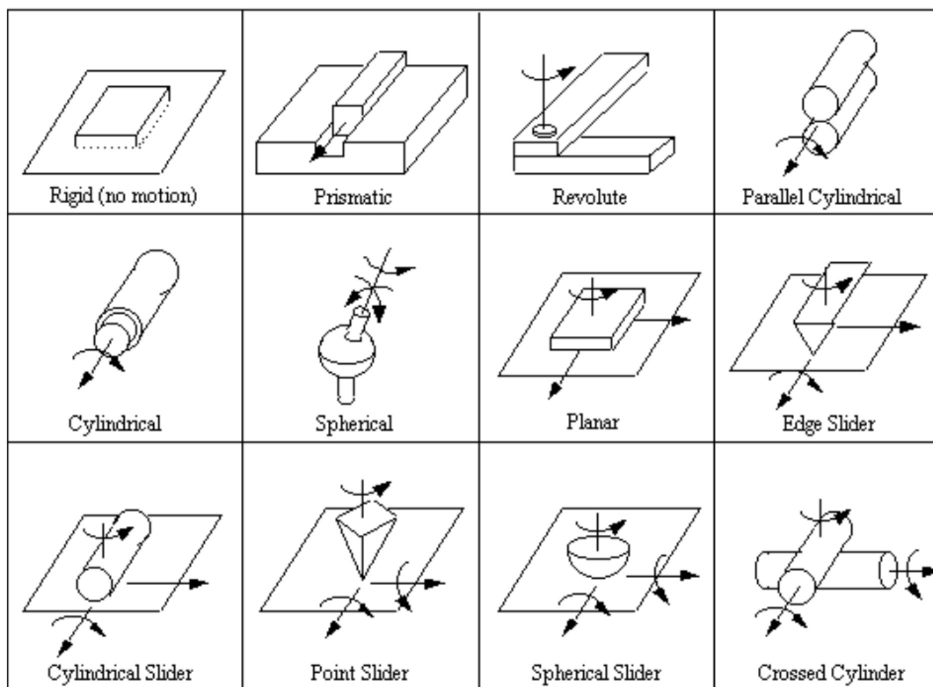


Abbildung 2.1: Verschiedene Typen von Gelenken mit null bis sechs DoF (Quelle: [Tou13a])

Freiheitsgrad (engl. Degrees of Freedom, DoF)

„Der Freiheitsgrad f ist die Anzahl der Koordinaten, die man mindestens braucht, um die Lage [. . .] eines mechanischen Systems eindeutig zu beschreiben.“ ([BSS07, S. 47]).

In der Robotik sind Gelenke und Arme als starre Körper modelliert. Je nach Art des repräsentierten Gelenks können zwei miteinander verbundene Körper zueinander eine Auslenkung mit einem oder mehreren Freiheitsgraden haben. Diese Beziehung wird für gewöhnlich *kinematisch* beschrieben. Die Summe der Freiheitsgrade ergibt den Freiheitsgrad des Systems, bzw. des Roboters. Abbildung 2.1 zeigt unterschiedliche Typen von Gelenken.

Mit dem Begriff *Freiheitsgrad* kann auch eine einzelne dieser Koordinaten gemeint sein, z.B. die Translation entlang einer Achse.

Pose
Die Pose beschreibt die Lage eines Körpers im Raum, bestehend aus dessen Position (Translation) und Orientierung (Rotation).

Systemkonfiguration
Bezeichnet eine bestimmte Pose des Systems bzw. Roboters, z.B. die Winkel aller Gelenke.

Endeffektor
Der *Endeffektor* bezeichnet den Körper des Roboters, dessen *Pose* man letztendlich steuern möchte. Im Falle eines Akkuschraubers ist z.B. das Ziel die Spitze des Schraubbits mit der richtigen Orientierung an die entsprechende Position im Schraubenkopf zu bringen. Diese Spitze ist also der Endeffektor.

Kinematik
Kinematik ist nach [Jaz10] eine geometrische Beschreibung der Bewegung eines Körpers anhand dessen *Pose* und deren Ableitungen über die Zeit.

Die *direkte Kinematik* bezeichnet das Problem, aus den lokalen Auslenkungen der einzelnen Gelenke die resultierende globale Pose eines Körpers, z.B. des *Endeffektors*, zu errechnen.

Ihr Gegenstück ist die *inverse Kinematik*, die sich mit der Frage beschäftigt, wie man für eine gegebene globale Pose eines Körper (des Endeffektors) die Systemkonfiguration, also die Auslenkungen der Gelenke, bekommt. Alle möglichen Systemkonfigurationen für diese Pose bezeichnen den *null space*.

Dynamik
Die Dynamik befasst sich mit dem wechselseitigen Zusammenhängen von Bewegung und Kräften wie der Trägheit oder der Schwerkraft.

Trajektorie
Bezeichnet den Pfad im Raum, entlang welchem sich ein Punkt (z.B. der Eneffektor) bewegt.

2.2 Generieren von Bewegungen

2.2.1 Problemstellung

Der erste Schritt bei der Generierung einer Bewegung ist das Aufstellen des Problems. Ein Problem besteht üblicherweise aus einem *Ziel* und *Randbedingungen* (engl. constraint), die die Bewegungsfreiheit eingrenzen.

Ein Ziel ist z.B. das Führen des Endeffektors zu einem bestimmten Punkt oder entlang einer gegebenen Trajektorie.

Randbedingungen können unterschiedlichster Art sein, unter anderem:

1. Dynamische Einschränkungen: z.B. die Trägheit und die Lage des Schwerpunkts zur Bewahrung des „Gleichgewichts“ oder das Einhalten der erlaubten Kräfte auf die Gelenke.
2. Einschränkungen durch die Umgebung: Die Vermeidung von Kollisionen mit Gegenständen oder Personen in der Umgebung können dazu führen, dass entweder von der direkten Trajektorie abgewichen werden muss oder einzelne Gelenke in ihrer Bewegungsfreiheit begrenzt sind.
3. Einschränkungen des Systems: mechanische Grenzen von Gelenken, Verhindern von Kollisionen mit dem eigenen Körper, etc.
4. Auflagen für globale oder lokale Posen:
    - Teile des Systems, die nicht bewegt werden dürfen (wie z.B. der Oberkörper)
    - die Ausrichtung des Endeffektors am Ziel (wie beim oben erwähnten Akkuschrauber)
    - die Orientierung von Fingerspitzen zueinander beim Greifen
    - globale Orientierung eines Körpers, z.B. eines gefüllten Glases

Hinzu kommen u.U. weiter Kriterien, wie z.B. die Dauer um die Bewegung auszuführen, die Einfluss etwa auf die Dynamik nehmen.

Das Problem muss dabei in einer exakten, maschinenlesbaren Form dargestellt werden.

2.2.2 Planung

Sind Ziele und Rahmenbedingungen bekannt, kann daraus eine Bewegung geplant werden. Für diese Aufgabe gibt es zahlreiche verschiedene Algorithmen. Das Kapitel 3.1.1 stellt einige dieser Verfahren vor.

Neben dem Bestreben, das Problem möglichst gut zu lösen, ist insbesondere auch die zum Finden einer adäquaten Lösung benötigte Zeit wichtig. Gerade in Umgebungen mit beweglichen Objekten, wie bei der Zusammenarbeit mit Menschen, muss auf die veränderte Situation so schnell reagiert, d.h. die geplante Bewegung abgeändert werden, dass die Rahmenbedingungen (z.B. Kollisionsvermeidung) weiterhin eingehalten werden.

Die Planung erfolgt im Allgemeinen im Zustandsraum (engl. *state space*). Bei einer kinematischen Betrachtung enthält dieser alle möglichen Systemkonfigurationen des Roboters, er wird dann auch oft als *configuration space* oder C-Space bezeichnet. Jeder Freiheitsgrad des Roboters entspricht dabei einer Dimension, der Zustandsraum eines Roboters mit 20 Freiheitsgraden ist demnach 20-dimensional.

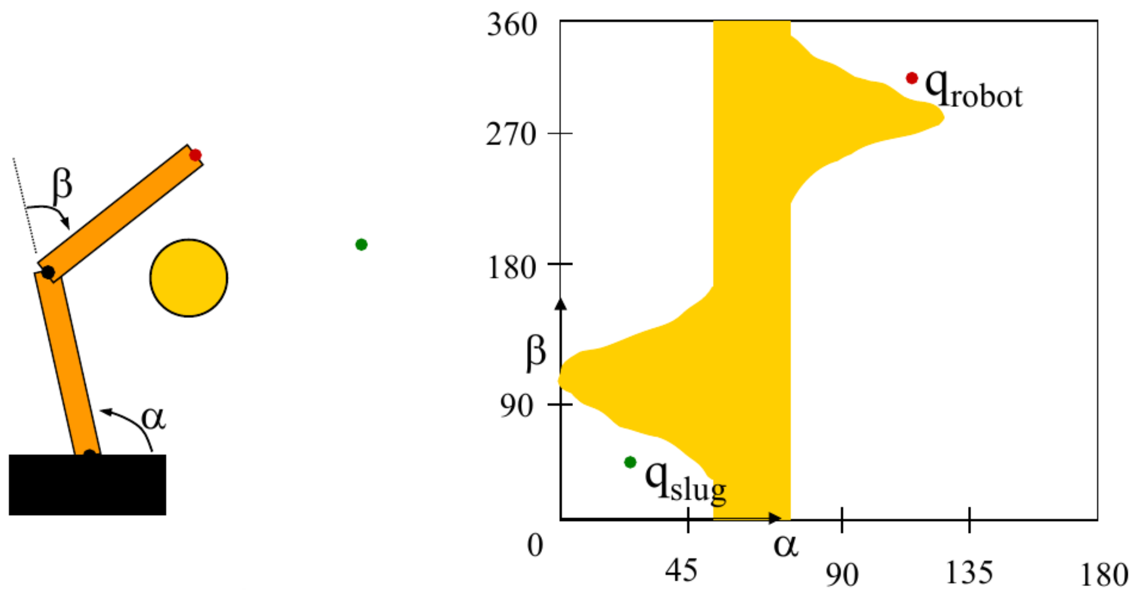


Abbildung 2.2: Ein primitiver Roboter mit zwei Freiheitsgraden und ein Hindernis und deren Repräsentation im ControlSpace (Quelle: [Hla10b, S. 40])

Sollen dynamische Aspekte in die Planung mit einfließen, muss auch die Ableitung der Konfiguration - die Geschwindigkeiten - mit einfließen. Für den oben erwähnten Roboter mit 20 Freiheitsgraden ist der Zustandsraum also 40-dimensional, da für jeden Freiheitsgrad Konfiguration und Ableitung enthalten sind.

Das Ziel eines Planers ist, in diesem Raum einen Pfad von einer Ausgangskonfiguration zu einer Zielkonfiguration zu erhalten. Dies kann beispielsweise durch eine Pfadsuche in einem abgeleiteten Graphen oder durch eine Optimierung erfolgen. Oft werden dabei auch Hindernisse vom euklidischen in den Zustandsraum übertragen, wie das auch in Abbildung 2.2 der Fall ist. Ein gültiger Pfad kann dann nur dort verlaufen, wo kein solches Hindernis liegt.

3 Stand der Technik

3.1 Related Work

3.1.1 Algorithmen zum Planen der Bewegung

Für die Planung der Bewegung existieren zahlreiche Algorithmen. Im Folgenden sind einige Methoden kurz vorgestellt.

Sample Based Motion Planning

Diese Methode ist nach [ZRD+13] im Bereich der Robotik sehr verbreitet, da sie mit recht hoher Wahrscheinlichkeit bei entsprechendem Aufwand eine Lösung findet.

Im freien Zustandsraum, also dort, wo kein Hindernis im Weg ist, wird eine bestimmte Menge an Punkten gesetzt, zum Beispiel zufällig im oder normalverteilt über den Raum. Aus diesen Punkten wird eine „Karte“ erstellt, wobei eine Verbindung zwischen zwei Punkten bedeutet, dass auf dem Weg zwischen ihnen kein Hindernis existiert. Schließlich kann eine Suche nach dem kürzesten Pfad auf diese Karte angewendet werden.

Der entstandene Pfad ist im Normalfall nicht optimal, kann aber, falls nötig, als guter Ausgangspunkt mit garantierter Lösung für ein Optimierungsverfahren genutzt werden.

Cell Decomposition

Bei der *cell decomposition* wird der komplette freie Zustandsraum in einzelne Zellen aufgeteilt, z.B. durch Trapezierung. Aus diesen Zellen wird dann ein Graph adjazenter Zellen erstellt, in dem ein Pfad von Anfangs- zu Ziel-Zelle gesucht wird, wie in Abbildung 3.1 zu sehen ist. Da innerhalb einer Zelle jede Konfiguration gültig ist, kann dort ein beliebiger Pfad gewählt werden.

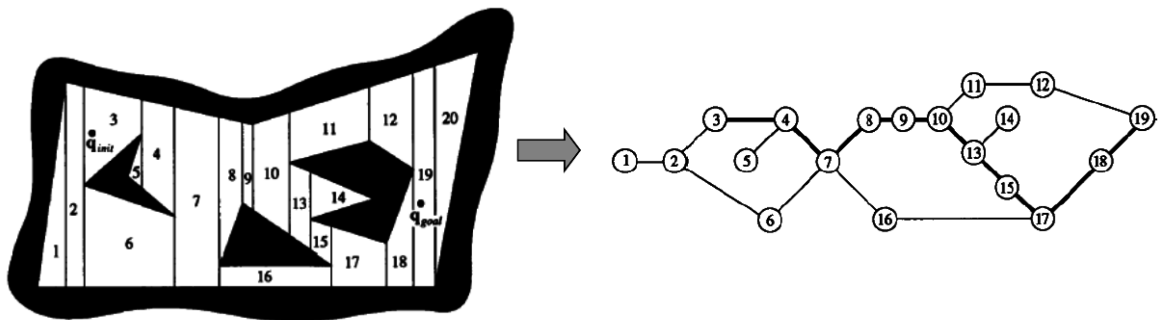


Abbildung 3.1: Ein Beispiel für Cell Decomposition: Der Zustandsraum wird in Trapeze aufgeteilt und im daraus enstehenden Adjazenzgraph nach einem Weg vom Start zum Ziel gesucht (Quelle: [Hla10a, S. 40]).

Potential Fields

Dieses Verfahren ist an Potentialfelder, wie z.B. Magnetfelder, aus der Physik angelehnt. Dabei hat das Ziel ein auf den Roboter anziehendes Potential, Hindernisse ein abstoßendes Potential. Die verschiedenen Potentiale üben eine vektorielle „Kraft" auf den Roboter aus, deren Summe den Roboter kontrollieren. Es kann dabei allerdings leicht dazu kommen, dass man in einem lokalen Minimum stecken bleibt, sodass das Ziel nicht erreicht wird.

Trajektorienoptimierung

Statt einer sonst üblichen Pfadsuche verfolgt man hier einen numerischen Ansatz. Die Kriterien des Problems sowie auch alle Hindernisse, werden in eine Kostenfunktion gefasst, die die Güte einer Trajektorie bewertet. Durch ein Optimierungsverfahren wird nun versucht, die Kosten zu minimieren. Oft werden die Optimierungsalgorithmen erweitert, um durch eine gewisse Ungenauigkeit oder das Zulassen von Verschlechterung zu vermeiden, in lokalen Minima stecken zu bleiben. Auch kann vor der Optimierung eine schnelle Pfadsuche ausgeführt werden und die gefundene Lösung als Ausgangspunkt verwendet werden.

Neben ORS (s. Kapitel 3.3) wird diese Methode z.B. auch von dem Algorithmus CHOMP ([ZRD+13]) verwendet.

Search Based

Die Programm-Bibliothek SBPL ([L+13]) verfolgt einen komplett anderen Ansatz. Statt eine komplett neue Bewegung zu suchen, wird eine Datenbank mit vorgefertigten Bewegungsbruchstücken zu Hilfe genommen, um durch Zusammensetzen einzelner Teile eine komplette Bewegung zu erstellen.

3.1.2 MoveIt!

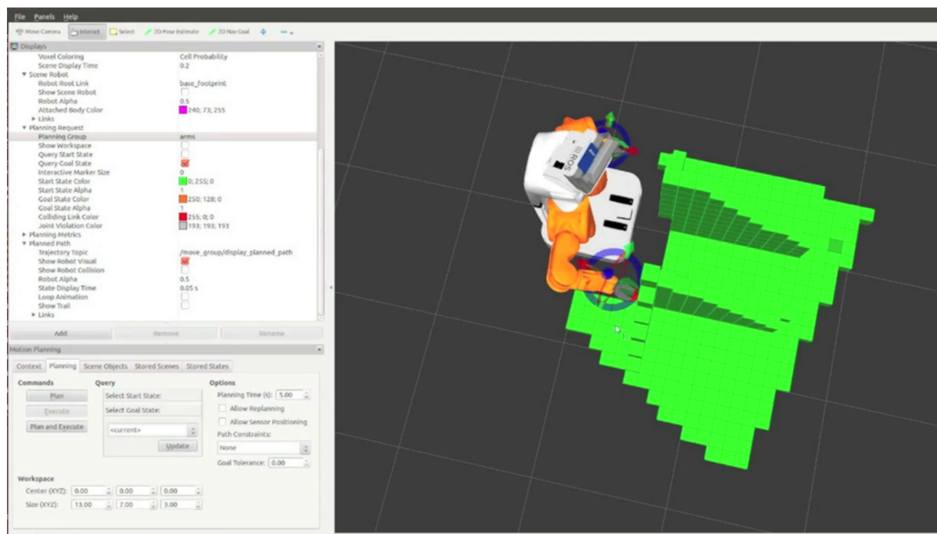


Abbildung 3.2: MoveIt! beim Erstellen einer Bewegung mit Darstellung der Wahrnehmung des PR2s (Quelle: [SC13])

Die Software MoveIt! ([SC13]) wurde im Mai 2013 von dem Unternehmen Willow Garage als „new software targeted at allowing you to build advanced applications integrating motion planning, kinematics, collision checking with grasping, manipulation, navigation, perception, and control“ ([Chi13]) vorgestellt. Sie ist kostenlos, quelloffen und dank BSD-Lizenz auch für die kommerzielle Nutzung geeignet.

Willow Garage wurde nach [Gar13] im Jahr 2006 mit dem Ziel gegründet, die Entwicklung von Robotern für zivile Zwecke und von Software im Bereich der Robotik zu beschleunigen. Es ist Erbauer des Roboters PR2 und des quelloffenen Robot Operating Systems ROS, das eine gemeinsame Basis für Robotik-Projekte bieten und damit die Wiederverwendbarkeit von Entwicklungen steigern soll ([ROS13]).

MoveIt! präsentiert sich als ein großes Programmpaket mit Funktionen von grafischer Benutzerschnittstelle über die Verwaltung von Planungsaufträgen bis hin zur Interaktion mit dem eigentlichen Roboter, wobei es sehr eng mit ROS zusammenarbeitet. Dabei stellt es hauptsächlich eine Rahmenanwendung dar, die an vielen Stellen durch Plugins erweitert werden kann, so zum Beispiel bei der Visualisierung von Vorgängen, der Kollisionserkennung oder der Planung.

Eine Besonderheit stellt sicher dar, dass die Wahrnehmungen des Roboters, z.B. durch Kameras oder Laserscanner, direkt angezeigt und auch in die Planung mit einbezogen werden können.

Außerdem stellt MoveIt! umfangreiche Benchmark-Werkzeuge zur Verfügung, über die zum Beispiel die verschiedenen Planer oder auch unterschiedliche Problemformulierungen objektiv miteinander verglichen werden können.

Zur Zeit wird MoveIt! mit drei verschiedenen Planern ausgeliefert:

- dem Optimierer CHOMP
- dem sample based „Open Motion Planing Library“ OMPL ([ŞMK12])
- dem suchbassierten SBPL

3.2 Gauss-Newton-basierter Optimierer

In [Tou13b] stellt Marc Toussaint ein Konzept vor, um Bewegungen mit Hilfe des numerischen Gauss-Newton-Verfahrens zu optimieren. Grundlage dafür ist die These, dass eben dieses Verfahren für die direkte Optimierung in hochdimensionalen Räumen, wie sie hier vorliegt, sehr effizient arbeite. Das Optimieren einer Trajektorie eines Roboters mit 20 Freiheitsgraden über 200 Zeitschritte sei im dynamischen Fall ein 8000-dimensionales Problem.

Alle Aspekte des Problems werden dabei als Teil einer vektoriellen Kostenfunktion ausgedrückt. Eine Besonderheit ist, dass auch die Kollisionsberechnung mit Hindernissen in der Welt als Bestandteil der Kostenfunktion modelliert wird. Mit der richtigen Gewichtung der Kosten stehe diese Methode der üblichen Methode von harten Beschränkungen, die den freien Zustandsraum begrenzen, im Resultat kaum nach. Das Vermeiden von harten Schranken führe dafür zu einer deutlichen Beschleunigung und Vereinfachung der Algorithmen.

3.2.1 Kosten

In diesem Abschnitt werden einige Kostenterme für konkrete Aufgaben im kinematischen Fall vorgestellt.

Es werden dabei die Kosten eines einzelnen diskreten Zeitschrittes betrachtet. Bei einer Folge von Zeitschritten muss entsprechend für jeden Schritt eine Kostenfunktion evaluiert werden. Die Jacobi-Matrizen für die einzelnen Terme sind dabei der Übersichtlichkeit halber weggelassen.

Notation

Für die im Folgenden vorgestellten Kostenterme wird die Notation aus [Tou13b] übernommen.

- q: die Systemkonfiguration.
- $x \in \mathbb{R}^n$: Zustand des Systems, also $x = q$ im kinematischen, $x = (q, \dot{q})$ im dynamischen Fall. x ist also ein Punkt im Zustandsraum.

- $c(x) = \phi(x)^T * \phi(x)$: Kostenfunktion über den Zustandsraum. $\phi : \mathbb{R}^n \rightarrow R^d$ beschreibt den „Kostenvektor“. Dieser enthält die Kostenterme der einzelnen Aspekte des Problems. Per Konvention beziehen sich die ersten Einträge auf Übergangskosten, dann folgen Einträge von den einzelnen Aufgaben (Task Costs).
- $J(x) = \partial_x \phi(x) \in \mathbb{R}_{d \times n}$: die Jacobi-Matrix zu ϕ. Es wird davon ausgegangen, dass sie stets für ein beliebiges x berechnet werden kann.
- $\|v\|_W^2 = v^T W v$: die quadratische Norm von v bezüglich der Metrik W. $\sqrt{W}$ bezeichnet die Cholesky-Zerlegung der Metrik W.
- $\varrho = \frac{1}{\sigma}$: ϱ bezeichnet die Präzision für eine Aufgabe, ihr Kehrwert σ kann oft als Standardabweichung betrachtet werden. Sie fließen als Gewichtung in die Terme ein.

Außerdem werden folgende Schreibweisen definiert:

- $\phi_{\mathrm{pos}iv}$: Die globale Position des Körpers i, verschoben um den Vektor v, entsprechend die Jakobi-Matrix $J_{\mathrm{pos}iv}$. Der Einfachkeit halber wird dabei im Folgenden auch von einem „Punkt“ (i, v) gesprochen.
- $\phi_{\mathrm{dir}iv}$: Die globale Ausrichtung des Körpers i mit der lokalen Ausrichtung v, entsprechend die Jakobi-Matrix $J_{\mathrm{dir}iv}$.
- R_i: Eine Rotationsmatrix, um lokale Koordinaten des Körpers i in Weltkoordinaten umzurechnen

Übergangskosten

Übergangskosten beschreiben die Kosten, welche beim Wechsel von einem (x) in einen anderen (x') Systemzustand entstehen. Der Kostenvektor ist daher für diese als $\phi(x, x')$ definiert.

In der Kinematik sollen große Änderungen im Systemzustand vermieden werden. Die einfachste Formulierung davon lautet

(3.1) $\phi(x, x') \leftarrow \sqrt{W}\left(q' - q\right)$

Soll eine „null space motion“ mit einbezogen werden, z.B. wenn sich die Weltkoordinaten des Roboters verändert haben (die relative Pose zu anderen wichtigen Objekten aber gleich bleibt), erweitert sich 3.1 um den Vektor a, den man auf den bisherigen Zustand aufaddiert:

(3.2) $\phi(x, x') \leftarrow \sqrt{W}\left(q' - (q + a)\right)$

Positionierung

Um den Punkt $\phi_{\mathrm{pos}iv}$ an eine gewünschte globale Position y^* zu bringen, werden Kosten für die Abweichung berechnet:

(3.3) $\phi(x, x') \leftarrow \left(\phi_{\mathrm{pos}iv}(q) - y^*\right) / \sigma$

σ kann als Standardabweichung von der gewünschten Position angesehen werden. Bei einer Skalierung des Koordinatensystems in Metern (also 1≙1m) entspräche $\sigma = 0.01$ einer Standardabweichung von 1cm.

Eine Positionierung eines Punkts (i, v) relativ zu einem anderen Punkt (j, w) erfolgt entsprechend:

(3.4) $\phi(x, x') \leftarrow \left(R_j^{-1}\left(\phi_{\mathrm{pos}iv}(q) - \phi_{\mathrm{pos}jw}(q)\right) - y^*\right) / \sigma$

R_j^{-1} überführt die Koordinaten in das Bezugsystem des Körpers j.

Ausrichtung

Die Ausrichtung eines Körpers ist ebenfalls eine häufige und wichtige Aufgabe, z.B. um ein gefülltes Gefäß aufrecht zu halten. Dafür nennt Toussaint zwei Varianten, das Vergleichen von Orientierungen und das Betrachten eines Skalarproduktes.

Ersteres ist im globalen Fall analog zu 3.3 unter Betrachtung von $\phi_{\mathrm{dir}iv}$ statt $\phi_{\mathrm{pos}iv}$:

(3.5) $\phi(x, x') \leftarrow (\phi_{\mathrm{dir}iv}(q) - y^*) / \sigma$

Wobei y^* die gewünschte Achse in Weltkoordinaten ist. Um die v-Achse des Körpers i also an der globalen z-Achse auszurichten gilt $y^* = (0, 0, 1)^\top$.

Um die relative Ausrichtung zu einem anderen Körper j zu betrachten, muss die Ausrichtung von (i, v) nur in das System von j rotiert werden:

(3.6) $\phi(x, x') \leftarrow \left(R_j^{-1}\phi_{\mathrm{dir}iv}(q) - y^*\right) / \sigma$

Bei dieser Notation kann σ als Standardabweichung im Bogenmaß betrachtet werden.

Die Betrachtung eines Skalarprodukts kann in Einzelfällen intuitiver sein. Insbesondere gilt das für eine Gleichrichtung (Skalarprodukt 1), entgegengesetzte Richtungen (Skalarprodukt -1) und Orthogonalität (Skalarprodukt 0). Sei v^* die gewünschte Achse, y^* das gewünschte Skalarprodukt, dann gilt:

(3.7) $\phi(x, x') \leftarrow \left(v^{*\top}\phi_{\mathrm{dir}iv}(q) - y^*\right) / \sigma$

Für den relativen Fall gilt $v^* = \phi_{\mathrm{dir}jw}(q)$, also:

(3.8) $\phi(x, x') \leftarrow \left(\phi_{\mathrm{dir}jw}^\top(q)\phi_{\mathrm{dir}iv}(q) - y^*\right) / \sigma$

3.3 OpenRobotSimulator ORS

ORS ist ein Softwarepaket zur Simulation von Robotern mit Schnittstellen für unterschiedliche Funktionen, entwickelt unter der Federführung von Marc Toussaint. Es besteht im Wesentlichen aus fünf Einheiten:

Core: Enthält grundlegende Funktionalitäten, die über mehrere Komponenten genutzt werden, wie Algorithmen und unterschiedliche Datenstrukturen. Darunter befinden sich der Datentyp `MT::Array`, der es erlaubt, Arrays beliebiger Dimensionalität und Typs mit dynamischer Größe zu erstellen und der `KeyValueGraph`, der Daten unterschiedlicher Typen in einer hierarchischen Struktur halten und textuell darstellen kann.

Ors: Enthält die Datenstruktur zur Repräsentation des Roboters und seiner Umgebung (den ORS-Graph) und damit zusammenhängende Funktionen. Eine genauere Beschreibung der einzelnen Komponenten ist im Kapitel 3.3.1 zu finden.

Gui: Kann einen Ors-Graph mit Hilfe von OpenGL anzeigen und bietet Schnittstellen zur Benutzerinteraktion an.

Optim: Enthält Optimierungs- und Evaluationsverfahren. Insbesondere ist dort auch der Optimierer nach dem Gauss-Newton-Verfahren implementiert.

Motion: Die Funktionalität und Datenstruktur für das Beschreiben und Generieren einer Bewegung mit Hilfe von Optimierung ist im Paket `Motion` enthalten. Dieses ist damit der primäre Interaktionspunkt, der in dieser Arbeit genutzt wird. In Kapitel 3.3.2 wird es näher beschrieben.

3.3.1 Der ORS-Graph

Der ORS-Graph ist die Grundlage aller Anwendungen, denn er beschreibt die simulierte Welt. Die Beschreibung besteht aus starren Körpern (Bodies), Formen (Shapes) und Gelenken (Joints). Außerdem kann noch die Annäherung von zwei Shapes als „Proxy“ (von englisch proximity) gespeichert werden, dies dient aber nur zu internen Zwecken und ist nicht Teil der Modellierung der Welt.

Wie in Kapitel 2.1 erläutert, besteht ein Roboter in der Beschreibung aus einzelnen starren Körpern. Jeder dieser Körper ist im ORS-Graph als Body beschrieben. Neben seiner aktuellen Pose und im aktuellen Systemzustand auf ihn wirkende Kräfte, kennt ein Körper auch Informationen über seine Masse, seine Trägheit und seinen Masseschwerpunkt. Außerdem kann er mehrere Shapes und Joints referenzieren, die Gestalt und Position im System definieren.

Über Shapes sind Ausprägungen in der Welt modelliert. Dadurch spielen sie bei der Berechnung von Abständen und der Kollisionsvermeidung eine wesentliche Rolle. Im Normalfall ist eine Shape immer einem Körper zugeordnet, annonyme Shapes können jedoch genutzt werden, dem Benutzer etwas anzuzeigen, was nicht direkt zur Welt gehört, wie etwa Markierungen. Neben verschiedenen primitiven Formen wie Quader, Zylinder oder Sphären kann

auch eine Gitternetzstruktur, z.B. aus einem CAD-Programm, oder eine Punktewolke, wie sie beispielsweise von Laserscannern erstellt wird, beschrieben werden.

Joints verbinden zwei Körper miteinander und beschreiben, wie sich diese zwei Körper zueinander bewegen können. Diese Beschreibung wird Teil der Systemkonfiguration. Durch eine Koordinatentransformation wird die genaue relative Position und Ausrichtung des Gelenks zu den Körpern angegeben.

Durch das Zusammenspiel dieser drei Objekttypen kann ein sehr genaues Modell des gewünschten Roboters erstellt werden. In Abbildung 5.1 auf Seite 35 ist ein darüber beschriebenes Modell des PR2 zu sehen.

3.3.2 Motion

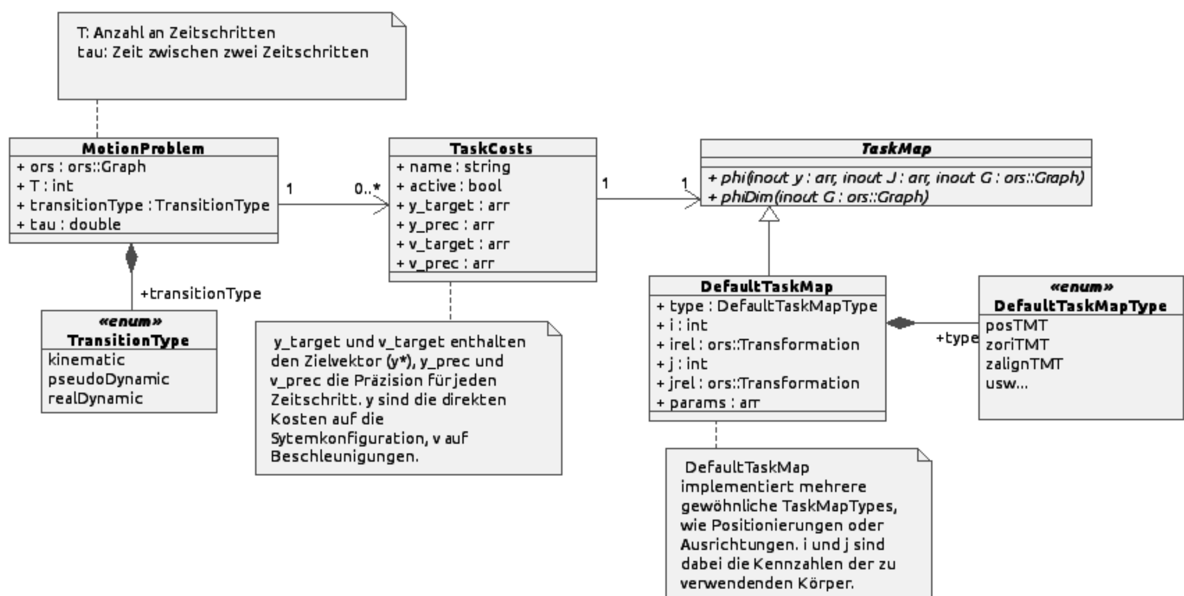


Abbildung 3.3: Vereinfachtes Klassendiagramm des ORS-Moduls `Motion`

Das Modul `Motion` leistet einen wichtigen Dienst in der praktischen Nutzbarkeit der Kostenterme, da es die Übergangs- und Task-Kosten kapselt und so dem Nutzer die Kenntnis der mathematischen Hintergründe abnimmt.

Ein `MotionProblem` bildet die Grundlage einer Problembeschreibung. Neben dem Ors-Graph, auf dem es arbeitet, enthält es den Übergangstyp, also ob z.B. ein kinematisches oder ein dynamisches Model genutzt werden soll, die Dauer der Trajektorie sowie die zu errechnende Anzahl von Zeitschritten und natürlich die definierten Aufgaben.

Die Aufgaben sind in zwei Elemente geteilt: Die `TaskMap`, die die ϕ-Funktion definiert und `TaskCosts`, die die Zielparameter (z.B. y^* bei einer Positionierung) sowie die Präzision (ϱ) über die Zeitschritte angibt.

Ein `TaskMap`-Objekt repräsentiert immer eine spezielle Aufgabe, einen `TaskMapType`, wie z.B. die Positionierung des Punktes (i, v). Es müssen also auch zum Beispiel Körper (i) und Transformationen (Translation v) konfiguriert werden. Welche Parameter eine TaskMap genau hat, hängt vom TaskMapType ab. Im Fall einer Ausrichtung anhand des Skalarprodukts, wie in Kapitel 3.2.1 vorgestellt, wird beispielsweise auch der Vektor v^* als Parameter übergeben.

Da die Kosten in der Regel nicht pro Zeitschritt einzeln angegeben werden sollen, bieten konkrete `TaskCosts` eine Schnittstelle, die die Kosten als Funktion über die Zeit definieren. So kann je nach Bedarf z.B. eine Spline-Interpolation oder eine intervallweise definierte Funktion gewählt werden. Die Trennung von der `TaskMap` erlaubt dabei, unabhängig voneinander neue Implementierungen hinzuzufügen.

Die Abbildung 3.3 zeigt ein Klassendiagramm mit den wichtigsten Eigenschaften der erwähnten Bestandteile.

4 Beschreibung von Bewegungsabläufen

Ein sehr wichtiger Aspekt ist die Möglichkeit, die Faktoren einer Bewegung akkurat und reproduzierbar zu beschreiben.

ORS bietet im Paket `Motion` eine Schnittstelle zu einer solchen Beschreibung in der Programmiersprache C++. Während diese Schnittstelle eine vollständige Beschreibung ermöglicht, ist es oft unpraktisch oder unerwünscht, das Problem direkt im Programmcode zu beschreiben. Insbesondere in Hinblick auf die Hauptaufgabe dieser Arbeit, dem Entwickeln einer grafischen Benutzeroberfläche zum Gestalten von Bewegungen (s. Kapitel 5 MotionDesigner) ist diese Möglichkeit nicht ausreichend, da sie nur statische Beschreibungen zum Zeitpunkt der Programmierung ermöglicht, jedoch keine dynamische Problemstellungen, wie es notwendig wäre.

Ein wesentlicher Punkt dieser Arbeit ist deshalb, ein Datenformat zu definieren, das es ermöglicht Beschreibungen zu speichern und dynamisch zu interpretieren. Dieses Datenformat soll die folgenden Eigenschaften erfüllen:

1. Die Beschreibung soll in sich abgeschlossen sein.

   Eine erstellte Beschreibung muss alle notwendigen Informationen enthalten, um sie auf einem anderen Gerät und/oder zu einem anderen Zeitpunkt wiederherstellen zu können. Sie darf deshalb keine Abhängigkeiten von der Arbeitsumgebung, wie globalen Einstellungen oder externen Dateien, haben.

2. Die Beschreibung soll in einem leicht lesbaren Format erfolgen, das ohne gesonderte Werkzeuge verwendbar ist.

   In der alltäglichen Arbeit mit ORS ist es oft notwendig, schnell Änderungen an einer Beschreibung vornehmen zu können. Deshalb ist es wichtig, dass diese in einem Format hinterlegt ist, das leicht und schnell zu editieren ist. Das bedeutet zum Einen, dass nach Möglichkeit keine speziellen Werkzeuge notwendig sein sollten, zum Anderen, dass die Darstellung für den Entwickler leicht zu verstehen und zu ändern ist.

3. Zukünftige Erweiterungen müssen mit geringem Aufwand realisiert werden können.

   ORS ist Gegenstand aktueller Forschungen. Das bedeutet, dass sich Struktur und Funktionsumfang in Zukunft stark verändern können. Das Format muss die Möglichkeit bieten, solche Veränderungen leicht in seiner Struktur zu repräsentieren.

4. Die Beschreibung muss robust gegen Fehler sein.

   Da die manuelle, direkte Manipulation der Beschreibung explizit vorgesehen ist, müssen syntaktische sowie semantische Fehler zu einem gewissen Grad toleriert werden können. Insbesondere bedeutet dies, dass lokal beschränkte Fehler (z.B. ein ungültiger Wert) keine globalen Auswirkungen haben sollten.

Das im Folgenden vorgestellte Format basiert auf dem von der ORS-Bibliothek definierten,KeyValueGraph, der bereits zur Beschreibung von Objekten in der Welt verwendet wird. Ein KeyValueGraph kann als gut lesbarer Text gespeichert und daraus wiederhergestellt werden.

Ein KeyValueGraph ist eine Liste von Elementen. Jedes Element besteht aus beliebig vielen Schlüssel-Worten, einer Liste von referenzierten Elementen und einem Wert. Ein Wert kann wiederum selbst ein KeyValueGraph sein, so dass sich eine hierarchische Struktur aufbauen lässt. In einem KeyValueGraph kann es mehrere Elemente mit gleichen Schlüsseln geben, diese sind also nicht eindeutig.

Durch die Benutzung von KeyValueGraphen muss sich der Entwickler keine weitere Syntax aneignen. Dies erlaubt ein schnelleres Einarbeiten und hilft, Fehler, die durch die Verwechslung der Formate entstehen, zu vermeiden. Außerdem kann bestehender, bereits durch andere Komponenten genutzter Code verwendet werden, was die Code-Basis klein hält.

4.1 Motion Problem Project

Die Beschreibung der Arbeitsumgebung erfolgt in einem "Motion Problem Project". Dieses umfasst

- Informationen zum Projekt, wie Name, Autor, Beschreibung
- Einstellungen der Trajektorie, wie Zeitschritte und Gesamtlänge
- Einstellungen des Optimizers, wie Abbruchbedingungen
- die genutzte Welt-Beschreibung (.ors-Datei)
- Definition von TaskMapTypes (Kapitel 4.3)
- Beschreibung der Tasks (Kapitel 4.2).

Die Beschreibung der Welt und Tasks sowie die Definition der TaskMapTypes kann jeweils sowohl integriert in einem untergeordneten KeyValueGraph oder in einer externen Datei erfolgen. Die Benutzung mehrerer Dateien erhöht die Übersichtlichkeit und ermöglicht dem Entwickler, die gesuchte Stelle schneller zu finden. Außerdem sind die Welt und die TaskMapTypes oftmals über lange Zeit und über viele gespeicherte Projekte konstant, sodass durch die Angabe einer gemeinsam genutzten Datei Redundanz vermieden werden kann.

Abbildung 4.1 zeigt die Datenhaltung des gesamten Projekts als ER-Diagramm.

Abbildung 4.1: Entity-Relationship-Diagramm der gesamten Projektbeschreibung

4.2 Task Beschreibung

Ein Task beschreibt ein bestimmtes Ziel der Bewegung. Ein typischer Task könnte sein: " [a] Betrachte die Position des Endeffektors relativ zum Zielpunkt. [b] Passe die Position des Endeffektors im Verlauf der Trajektorie der des Ziels an, sodass die Positionen am Ende identisch sind. [c] Eine Standardabweichung von 1cm soll eingehalten werden. "

Solch ein Ziel besteht aus zwei Komponenten: der TaskMap (*a*) und den TaskCosts (*b* und *c*). Darüber hinaus besitzt ein Task noch Metainformationen wie eine Bezeichnung oder Aktivität.

4.2.1 TaskMap

Die TaskMap beschreibt die Situation, die betrachtet werden soll. Dies kann, wie im obigen Beispiel, eine Position sein, die Ausrichtung eines Arms, die Auslenkung der Gelenke, usw.

Wie in Kapitel 3.2.1 erläutert, werden Aufgaben der Problemstellung in ORS durch Kostenfunktionen angegeben. Diese Kostenfunktionen sind sehr abstrakt und schwer zu le-

Listing 4.1 Beschreibung eines Tasks bestehend aus Metadaten, TaskMap und TaskCosts

```
1  task {
2        name='Move to target'
3        activated=true
4
5        map posTMT {
6              iBody=endeff
7              jBody=target
8        }
9
10       costs pos spline {
11             degree=1
12             target=[0 0 0]
13             prec=100
14       }
15 }
```

sen, weshalb sie abstrahiert werden. Eine TaskMap ist die Repräsentation einer bestimmten Kostenfunktion (vgl. TaskMapType) und den darin verwendeten spezifischen Parametern. Wenngleich eine beliebige Menge unterschiedlicher Kostenfunktion denkbar ist, gibt es eine begrenzte Anzahl an sinnvoll zu verwendenden Aufgaben. Daher ist eine Abstraktion dieser Kostenfunktionen sinnvoll, fördert die Benutzbarkeit und vermeidet Fehler, die durch die direkte Nutzung der Funktionen entstehen könnten.

Die Aufgabe einer TaskMap ist es, technisch betrachtet, den Kostenvektor $\phi(x)$ für die Problemstellung zu erstellen. Sie bietet dafür eine Schnittstelle auf die benutzerdefinierten Parameter an.

Die textuelle Beschreibung beginnt mit dem Schlüsselwort `map` gefolgt vom Bezeichner des TaskMapTypes (Kapitel 4.3) und, in geschweiften Klammern, den Parametern des TaskMapTypes als KeyValueGraph.

Listing 4.1 zeigt in Zeilen 5-8 die TaskMap vom Typ `posTMT` für das obige Beispiel. Der Parameter `iBody` gibt den Namen des aktiven Körpers (Endeffektor) an, `jBody` das optionale Bezugssystem (Ziel).

4.2.2 TaskCosts

Die TaskCosts beschreiben zum einen den idealen Pfad der betrachteten Situation über die Zeitschritte hinweg (b) (bspw. den Abstand zwischen zwei Objekten), zum anderen eine Präzision, mit der dieser Pfad eingehalten werden soll. Dabei entspricht der Kehrwert der Präzision per Konvention der gewünschten Standardabweichung in Metern (vgl. [Tou13b]).

Um von der Anzahl der Zeitschritte der Trajektorie unabhängig zu bleiben und um die Lesbarkeit zu erhöhen, werden TaskCosts über den Bezeichner einer Funktion sowie dessen Parameter definiert. Im MotionDesigner werden zur Zeit B-Splines beliebigen Grades genutzt. Als Parameter werden die De-Boor-Punkte für Pfad und Präzision angegeben.

Listing 4.2 Beschreibung eines TaskMapTypes

```
taskMapType posTMT {
   class=DefaultTaskMap

   param string iBody={
      optional=false
      defaultValue=endeff
      ui=BodyList
   }

   param string jBody={
      optional
      ui=BodyList
   }
}
```

Neben positionsbezogenen TaskCosts können für einen Task auch beschleunigungsbezogene Kosten angegeben werden.

Die textuelle Repräsentation beginnt mit dem Schlüsselwort `costs` gefolgt vom Typ der Kosten (`pos` für positionsbezogene, `vel` für beschleunigungsbezogene Kosten) und dem Funktionsbezeichner.

In Listing 4.1 werden in Zeilen 10-14 die Kosten aus obigem Beispiel als Spline 1. Grades (Zeile 11) beschrieben. Es wird lediglich ein Punkt, das relativen Ziel $\vec{0}$ (Zeile 12) und die Präzision 100 (Zeile 13), angegeben, da die Realisierung der Splines-Funktion automatisch die aktuelle Startposition als ersten Punkt hinzufügt.

4.3 TaskMapType Beschreibung

Um, wie zuvor als Kriterium für das Dateisystem angegeben, auf zukünftige Änderungen an ORS reagieren zu können, werden durch TaskMapTypes mögliche TaskMaps auf einer Metaebene deklariert. Damit haben die TaskMapTypes eine eher technisch orientierte Aufgabe. Sie definieren, durch welche Klasse die TaskMap realisiert wird und die Parameter, die die Map verwendet.

In der Programmierung können diese Informationen verwendet werden um beispielsweise automatisch eine Benutzerschnittstelle zu generieren. Tatsächlich wird in MotionDesigner sowohl die grafische Oberfläche (Kapitel 5.3 und 5.4.2) als auch das Model (Kapitel 5.4.1), die programminterne Repräsentation des Problems, dynamisch aus den TaskMapTypes generiert.

Der praktische Vorteil ist, dass beim Ändern oder Hinzufügen von neuen TaskMaps keine Anpassungen in bereits existierenden Programmen vorgenommen werden müssen, sondern nur in den TaskMapTypes.

Listing 4.2 zeigt beispielhaft die Definition für die in Listing 4.1 verwendete TaskMap `postTMT`.

Ein TaskMapType wird über das Schlüsselwort `taskMapType`, gefolgt von einem eindeutigen Namen definiert (Zeile 1). Für diesen Typ muss mindestens die realisierende Klasse angegeben werden (Zeile 2). Außerdem werden die Parameter der Map definiert (Zeilen 4-8 und 10-13).

`param` leitet die Definition eines Parameters ein, gefolgt von einem Typen und dem Namen des Parameters. Jeder Parameter kann seinerseits wieder eigene Eigenschaften besitzen. Standardmäßig kann angegeben werden, ob der Parameter optional ist (Zeile 5 bzw. 11, die Notation `optional` in Zeile 11 ist die Kurzform für `optional=true`), mit welchem Standardwert der Parameter vorbelegt sein soll (Zeile 6) und Informationen für die Generierung einer graphischen Oberfläche (Zeile 7 bzw. 12). In diesem Fall wird über „BodyList“ angegeben, dass keine freie Eingabe eines Strings erfolgen, sondern eine Auswahlliste aller in der ORS-Datei namentlich definierten Körper angezeigt werden soll. Außerdem können je nach Typ noch weitere Eigenschaften vorhanden sein. Der Tensor-Typ `arr` nimmt zum Beispiel Informationen über Anzahl und Größe der Dimensionen an, sowie darüber, ob Anzahl und Größe konstant sind oder zur Laufzeit verändert werden können.

Nach Evaluation der in ORS definierten TaskMapTypes und deren Parameter wurden folgende Parametertypen implementiert.

bool Ein boolscher Wert.

string Eine Zeichenkette.

int Eine ganze Zahl.

double Eine Fließkommazahl.

intlist Eine Liste von ganzen Zahlen, durch Leerzeichen getrennt.

stringlist Eine Liste von Zeichenketten, jede Zeichenkette ist in Hochkommata eingeschlossen, mehrere Zeichenketten sind durch Kommata getrennt.

arr Ein Parameter des ORS-Array-Typs für Fließkommazahlen. Dieser erlaubt das Speichern von Tensoren beliebiger Dimensionalität und Größe.

Dieser Parametertyp besitzt weitere Eigenschaften:

dims=<int|array> Wenn der Wert eine ganze Zahl ist, spezifiziert es die Anzahl an Dimensionen, wenn ein Array von ganzen Zahlen angegeben ist (Format: `[i j k..]`), entspricht es den Dimensionsgrößen des Tensors.
Beispiel: Mit `dims=[4 3 2]` wird standardmäßig ein 4x3x2-Tensor erstellt.

dimNumEditable=<bool> Wenn wahr, darf die Anzahl der Dimensionen verändert werden, sonst ist die in `dims` angegebene Dimensionalität fest. Nur in Verbindung mit `dims`, sonst immer wahr.

dimSizeEditable=<bool> Wenn wahr, darf die Größe einzelner Dimensionen verändert werden, sonst ist die Größe fest. Nur in Verbindung mit `dims=<array>`, da sonst die Größe jeder Dimension 0 ist.

transformation Eine Koordinatentransformation. Sie wird als String-Liste einzelner Transformationsschritte gespeichert, ist aber auf Grund der Implementierung von einer `stringlist` unterschieden.

item Ein kombinierter Parameter, bestehend aus einem Körper des ORS-Models (als String) und einer Transformation.

5 MotionDesigner

In diesem Kapitel möchte ich den praktischen Teil meiner Arbeit, das Programm MotionDesigner, vorstellen.

Zunächst möchte ich in Kapitel 5.1 darstellen, was das Ziel des Programms sein soll und welche Motivation dahinter steht. In Kapitel 5.2 ist erläutert, welcher Nutzerkreis mit MotionDesigner angesprochen werden soll.

In Kapitel 5.3 wird schließlich die Oberfläche von MotionDesigner vorgestellt, gefolgt von einigen Aspekten der Implementierung in Kapitel 5.4.

5.1 Motivation

Eine Bewegung wird von einem Menschen meist intuitiv ausgeführt. Man weiß aus Erfahrung, wie eine Bewegung ausgeführt werden muss, um den Auflagen wie „berühre keine unbeteiligten Gegenstände“ oder ähnlichem zu entsprechen. Man erreicht sein Ziel, ohne viel darüber nachdenken zu müssen, wie der Ablauf der Bewegung im Einzelnen aussieht und auf welche Randbedingungen dabei geachtet werden muss. Einen solchen Ablauf akkurat zu beschreiben, ist daher oft sehr schwierig und es werden notwendige Details vergessen.

ORS besitzt dieses menschliche Erfahrungswissen nicht. Soll der in ORS simulierte Roboter eine Bewegung wie gewünscht durchführen, ist eine solche strikte Beschreibung unbedingt notwendig. Jedes vergessene Detail kann zu einer drastischen Abweichung des Resultats führen.

Zum Entwerfen einer hochwertigen Bewegung ist deshalb die direkte Rückmeldung der Wirkung kleiner Anpassung sehr hilfreich, da direkt sichtbar wird, ob eine Komponente des Problems vergessen wurde oder ungenau ist. ORS bietet die Möglichkeit, eine dreidimensionale Animation einer Trajektorie anzuzeigen. Das übliche Vorgehen zu Beginn dieser Arbeit ist, die Problemstellung durch die Schnittstellen in C++ zu beschreiben, das Programm zu kompilieren und auszuführen, um das Ergebnis der Beschreibung zu kontrollieren. Anpassungen erfolgen quasi „blind“ durch kleine Änderungen im Quelltext, gefolgt vom erneuten Kompilieren und Ausführen. Daraus resultieren viele Iterationen mit minimalen Änderungen, die die Effizienz sehr begrenzen.

Zusätzlich ist die Gestaltung der Bewegung durch Programmcode schwer zu erlernen und fehleranfällig. Da die Programmierschnittstelle oftmals abstrakte Werte verlangt, wie zum Beispiel die ID eines Körpers, muss der Benutzer eine sehr gute Kenntnis der Bedeutung einzelner Funktionsparameter haben. Das bedeutet, dass er sich zunächst intensiv mit der

Schnittstelle auseinandersetzen muss. Hinzu kommt, dass er Kenntnisse der Programmiersprache C++ haben muss. Der Benutzung von ORS steht aktuell also eine steile Lernkurve entgegen. Wird das Projekt nach einiger Zeit wiederverwendet, oder möchte ein anderer Benutzer ein bestehendes Projekt verwenden, ist durch die schlechte Lesbarkeit eine lange Einarbeitungszeit nötig, um die Aufgaben und Funktion zu begreifen.

Diese Problematik soll durch eine graphische Benutzerschnittstelle (GUI) verbessert werden. Eine grafische Schnittstelle zur Erstellung einzelner Tasks unter unterschiedlich stark ausgeprägter Hilfeleistung des Programms soll eine Nutzung ohne Kenntnisse der Implementierung oder der Programmiersprache ermöglichen und helfen, sich in bestehende Projekte schneller einarbeiten zu können. Ziel ist es weiterhin, dass der Anwender in Echtzeit Rückmeldung zur Auswirkung einer getätigte Änderungen erhält. Zusätzliche Visualisierungen der Kosten, also der Abweichung vom Idealverlauf, sollen neben der subjektiven, visuellen, eine objektive Bewertung der berechneten Trajektorie ermöglichen.

5.2 Zielgruppe

Wenngleich das Programm die nötigen Vorkenntnisse des Nutzers deutlich verringern soll, zielt es doch auf Anwender mit wissenschaftlichem Hintergrund im Bereich Robotik ab. Insbesondere ist es nötig, die Konzepte hinter der Generierung von Bewegung als Optimierungsproblem, auf denen ORS basiert, zu kennen und die resultierenden Konsequenzen zu verstehen.

Die primäre Zielgruppe wird deswegen in Personen gesehen, welche sich zum aktuellen Zeitpunkt und in Zukunft mit der Entwicklung von ORS beschäftigen oder ORS als Plattform zur Forschung im Bereich Motion Generation nutzen. Hier möchte MotionDesigner als hilfreiches Werkzeug zur Seite stehen und die Arbeit erleichtern und beschleunigen.

5.3 Benutzeroberfläche

5.3.1 Allgemeiner Aufbau

Die grafische Benutzeroberfläche von MotionDesigner teilt sich in vier Bereiche.

Auf der linken Seite befindet sich eine Liste aller Tasks des aktuellen Problems. Den Hauptbereich in der Mitte nimmt die Detaildarstellung eines Tasks ein. In diesem Bereich werden alle Änderungen an dem aktuell in der Liste ausgewählten Task durchgeführt.

Auf der rechten Seite befinden sich standardmäßig Informationen über die resultierende Trajektorie: Graphen zur Darstellung der Kosten und dem Verlauf des Optimierungsprozesses sowie die 3D-Animation der optimierten Bewegung. Unterhalb des Hauptbereichs kann über eine Konsole die Arbeit der ORS-Bibliothek verfolgt werden, um Entwicklern nähere Informationen zum Ablauf aus erster Hand zu geben. Diese sogenannten „Docks“ können vom

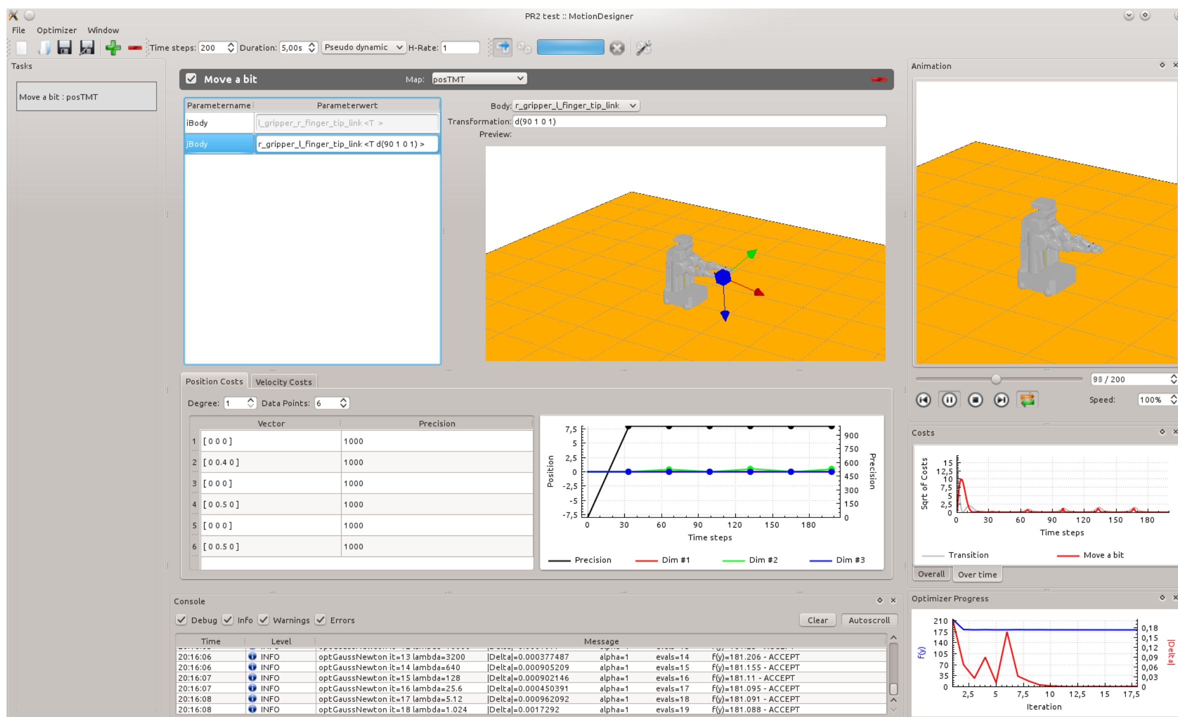


Abbildung 5.1: MotionDesigner Hauptfenster mit einem klatschenden PR2

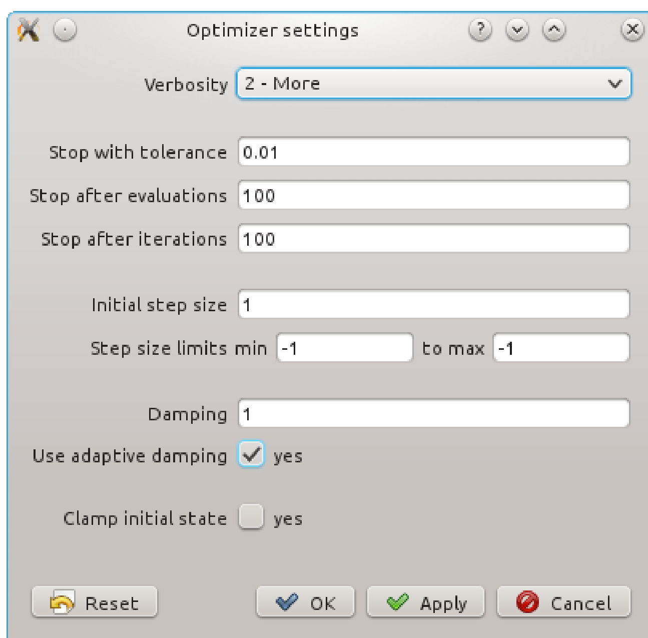


Abbildung 5.2: Einstellungen des Optimierers

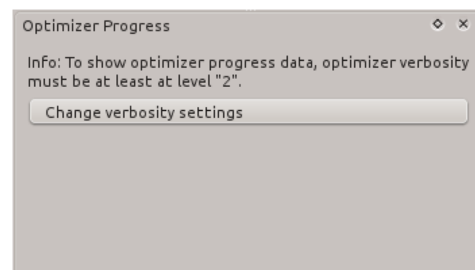


Abbildung 5.3: Für die Zwischenstandsanzeige des Optimierers muss die „Gesprächigkeit“ erhöht werden.

Anwender beliebig ein- oder ausgeblendet und an den Kanten des Programms angeordnet werden. Außerdem können einzelne Docks aus dem Fenster herausgelöst werden, um sie z.B. auf einen zweiten Monitor zu verschieben.

In der Toolbar lassen sich Einstellungen zur Bewegung vornehmen, wie die Anzahl der Zeitschritte, die Länge der Trajektorie und den Transitionstyp. Über die Menüführung lässt sich ein Dialog mit Einstellungen zum Optimierer öffnen (Abbildung 5.2). Damit auch verschiedene Einstellungen direkt ausprobiert werden können, ist dieser so gestaltet, dass er das Hauptfenster nicht blockiert sondern parallel in beiden Fenstern gearbeitet werden kann.

Wie in Kapitel 4 erwähnt, ist das Dateiformat in Projekten organisiert. Es ist also zunächst nötig, ein neues Projekt zu erstellen oder ein bestehendes Projekt zu öffnen. Beim Erstellen eines Projekts muss neben Speicherort auch das ORS-Modell und die zu verwendende TaskMapType-Definition ausgewählt werden. Dabei wird man durch einen Dialog unterstützt.

Innerhalb eines Projektes kann man über die Toolbar oder das Kontextmenü der Taskliste einzelne Tasks hinzufügen oder entfernen. Um einen Task zu bearbeiten, wird dieser in der Liste ausgewählt. Der Hauptbereich zeigt dann alle Informationen zum Task an.

Der Hauptbereich teilt sich vertikal in drei Teile. Zu oberst befindet sich die Titelleiste, über die der Task aktiviert bzw. deaktiviert werden kann, der Name angezeigt wird und, durch Klick auf denselben, auch geändert werden kann. Außerdem wird hier die TaskMap für diesen Task ausgewählt.

Darunter werden die Parameter für die gewählte TaskMap angezeigt. Auf der linken Seite gibt eine tabellarische Ansicht einen Überblick über alle Parameter. Die Werte der Parameter können, soweit es der Parametertyp zulässt, direkt in dieser Tabelle editiert werden. Dabei sind die Felder zum Schutz vor versehentlichen Manipulationen deaktiviert und müssen erst durch Klick auf die Zeile aktiviert werden.

Komplexere Feldtypen bieten bei Auswahl der Zeile rechts neben der Tabelle eine Detailansicht, zum Beispiel für eine Visualisierung oder einen ausführlicheren Editor. Im Abschnitt 5.3.2 werden die verschiedenen Parametereditoren vorgestellt.

Im unteren Bereich der Hauptansicht befindet sich der Editor für die Kosten. Dieser teilt sich wiederum in eine tabellarische Ansicht links und einen graphischen Editor rechts. In der aktuellen Version werden die Kosten über B-Splines beliebigen Grades interpoliert. Zur erleichterten Handhabung ermöglicht der Graph rechts das direkte Anpassen der Koordinaten der De-Boor-Punkte. Dabei wird in Echtzeit die resultierende Kurve angezeigt.

Jeder Task kann sowohl positionsbezogene als auch beschleunigungsbezogene Kosten besitzen. Zwischen ihnen kann über die Reiter oberhalb der Tabelle gewechselt werden.

Standardmäßig beginnt die Optimierung nach einer Änderung automatisch, um dem Benutzer schnellst möglich und ohne eigenes Zutun Rückmeldung über die Auswirkung zu geben. Dieses Verhalten kann bei Bedarf über die Menüstruktur deaktiviert und die Optimierung manuell gestartet werden. Über den Zustand des Optimierers gibt eine Fortschrittsanzeige in der Toolbar Auskunft. Während des Optimierungsvorgangs zeichnet das Dock „Optimizer

Progress“ live einen Graphen von den Zwischenergebnissen der Kosten und der Differenz zum vorhergehenden Durchlauf. Um diese Informationen zu erhalten, muss die „Verbosity“, also, zu welchem Detailgrad das aktuelle Geschehen mitgeteilt wird, des Optimierers mindestens auf Level 2 stehen. Ist dies nicht der Fall, bietet das Dock an, die Einstellung automatisch durchzuführen (Abbildung 5.3).

Nachdem die Optimierung abgeschlossen ist, kann die errechnete Trajektorie im Dock „Animation“ betrachtet werden. Die Steuerung erlaubt den einmaligen oder kontinuierlichen Durchlauf der Animation oder auch das Betrachten eines einzelnen Zeitschritts, um die Veränderungen im Detail zu bewerten. Die Abspielgeschwindigkeit kann dabei an den aktuellen Bedarf angepasst werden.

Das Dock „Costs“ enthält zwei unterschiedliche Ansichten der Transitions- und Taskkosten. Der Reiter „Overall“ zeigt die Summe der Kosten über alle Zeitschritte hinweg als Balkendiagramm an und fasst dazu noch die Kosten aller Tasks zusammen. Auf der Seite „Over time“ sind die Kosten pro Task/Transition und Zeitschritt aufgetragen, um problematische Zeitabschnitte zu identifizieren.

5.3.2 Parametereditoren

Standardeditor

Für alle Parametertypen ist ein Standardeditor definiert. Dieser zeigt in der Tabelle ein einfaches Textfeld mit der KeyValueGraph-Repräsentation des Parameterwertes. Dadurch ist auch bei einer zukünftigen Erweiterung der Parameterbeschreibung um weitere Typen immer ein Standardeditor verfügbar.

Da der Editor keine Informationen über den Typ besitzt, findet hierbei keine direkte Validierung der Benutzereingaben statt. Kann der dargestellte Parameter mit der Benutzereingabe nicht umgehen, wird die Änderung nicht in die Datenhaltung übernommen.

Dieser Editor verfügt über keine Detailansicht.

Checkbox

Für Parameter des Typs `bool` ist der Standardeditor überschrieben. Hier wird eine einfache Auswahlbox angezeigt.

Dieser Editor verfügt über keine Detailansicht.

Auswahl eines ORS-Bodys

Wenn ein Parameter vom Typ `string` den Namen eines ORS-Bodys enthalten soll, kann in der TaskMapType-Definition dieser Editor angegeben werden. Er stellt eine Liste aller namentlich definierten Bodys im ORS-Model zur Auswahl. Wenn der Parameter optional ist, kann der Benutzer auch keine Auswahl treffen.

Dieser Editor verfügt ebenfalls über keine Detailansicht.

Items

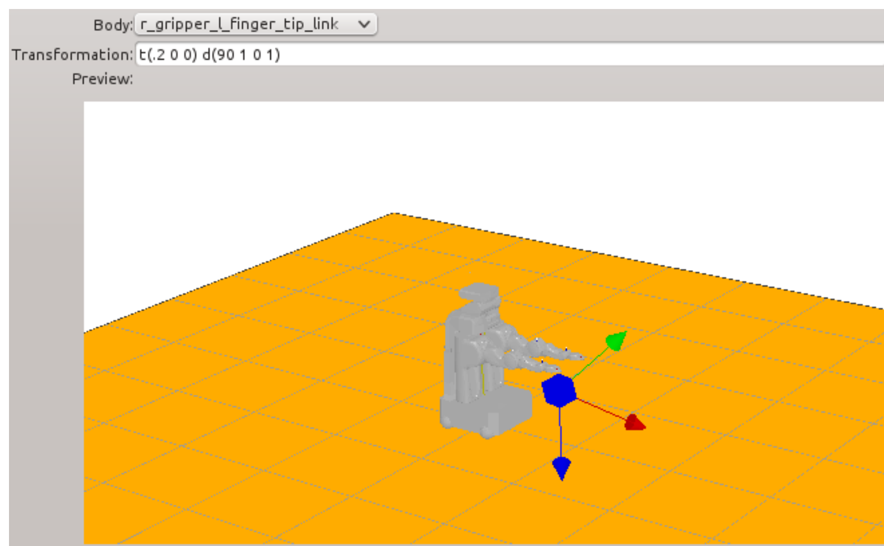


Abbildung 5.4: Detailansicht eines Item-Parameters

Parameter des Typs `item` zeigen in der Tabelle den Name des ausgewählten Körpers und die angewendete Transformation als Text an. Die Zelle ist für direktes Editieren gesperrt.

In der Detailansicht erfolgt die Auswahl des Bodies über ein DropDown-Feld und die Angabe der auf ihn anzuwendenden Transformation über ein Textfeld. Dabei wird die Eingabe validiert und nur gültige Transformationsschritte übernommen.

Die Auswirkung der Transformation auf den Körper wird in einer 3D-Darstellung bereits während der Eingabe visualisiert.

Liste von Shapes

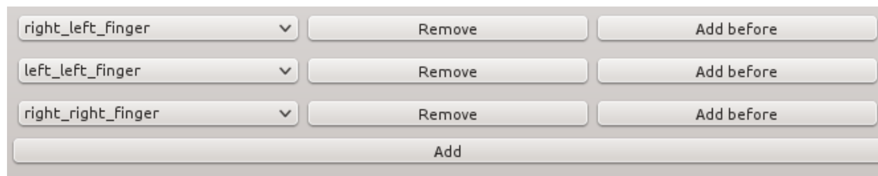
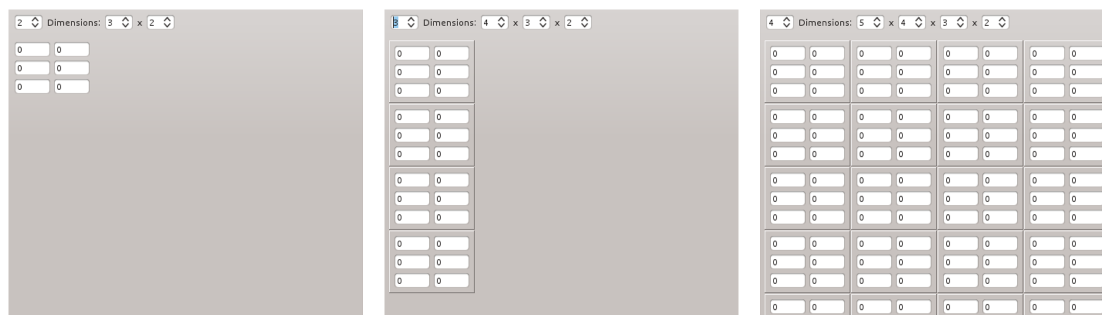


Abbildung 5.5: Detailansicht für eine Liste von Shapes

Wenn die Namen mehrerer Shapes angegeben werden sollen, wird der Benutzer über diesen Detail-Editor unterstützt. Er ermöglicht es, eine beliebige Menge von Shapes über Dropdowns in sortierter Weise auszuwählen. Wenn der Parameter optional ist, kann die Liste auch leer sein, sonst kann die letzte vorhandene Zeile nicht gelöscht werden.

Tensoren



(a) Tensor 2. Stufe **(b)** Tensor 3. Stufe **(c)** Tensor 4. Stufe

Abbildung 5.6: Detailansicht für Tensoren

Da Parameter des Typs `arr` eine beliebig große Dimensionalität besitzen können, ist dieser Detaileditor darauf ausgelegt, all diese Tensoren auch anzeigen zu können.

Das Tabellenfeld stellt den Tensor als eine Reihe von Fließkommazahlen nach der in [Knu08, S. 296] erläuterten Row-Major-Ordnung dar. Diese Darstellung erlaubt bei geringer Komplexität eine schnelle Eingabe, wird mit wachsender Größe des Tensors aber sehr schnell unübersichtlich.

Der Detaileditor zeigt jeweils zwei Dimensionen als Matrix dar, wobei jedes Element der Matrix die untergeordnete Matrix enthält. Matrizen werden dabei immer durch Rahmen zusammengefasst. Bei ungerader Dimensionszahl wird eine vertikale Darstellung („Vektor“) einer horizontalen vorgezogen (vgl. Abbildung 5.6b). Mindestens bis zu Tensoren vierter Stufe bietet die Darstellung eine hinreichende Übersichtlichkeit, wie in Abbildung 5.6 zu sehen ist.

5.4 Implementierung

MotionDesigner wurde in C++ nach den Grundsätzen der Objektorientierung implementiert.

Das Ziel der Objektorientierung ist nach [LL10], Bestandteile einer fachlichen Domäne durch Klassen zu repräsentieren. Ein wichtiger Aspekt ist dabei, alle für die Anwendung unwichtigen Entwurfsentscheidungen, wie z.B. genutzte Algorithmen oder die intern verwendete Speicherstruktur, nach außen hin zu verstecken, um eine möglichst kleine und klare Schnittstelle anzubieten. Eine Konsequenz dieses Information Hidings ist, dass externe Zugriffe auf die Daten ausschließlich über Methoden (sog. Getter- und Setter-Methoden) erfolgen, statt direkt über die entsprechenden Variablen. Dadurch kann die Klasse auch Maßnahmen ergreifen, die beispielsweise die Datenintegrität sicherstellen, wie zum Beispiel das Realisieren von Algorithmen zur Absicherung bei nebenläufigen Zugriffen.

Da für die Realisierung der grafischen Oberfläche das Framework Qt[1] zum Einsatz kommt, wurde die darin verwendete, vom Entwurfsmuster Model-View-Controller (kurz: MVC) abgeleitete, Model-View-Architektur übernommen.

Das aus der Programmiersprache Smalltalk stammende MVC-Pattern wird in [GHJV95, S. 4] so beschrieben:
„MVC consists of three kinds of objects. The Model is the application object, the View is its screen presentation, and the Controller defines the way the user interface reacts to user input.“
Dabei sind Model und View durch ein Benachrichtigungssystem verbunden, über das das Model Änderungen propagiert. Die View hat die Aufgabe, ihre Darstellung daraufhin anzupassen. Mehrere Views können das gleiche Model verwenden.

Qt verschmilzt die Ebenen View und Controller zu einer, um deren Komplexität zu veringern ([Dig13a]). Die Trennung zwischen der Datenhaltung (Model) und der Benutzerschnitstelle (View) bleibt weiterhin vorhanden.

Im Folgenden möchte ich auf einige ausgewählten Entwurfsentscheidungen und Umsetzungen eingehen.

5.4.1 Model

Das Model von MotionDesigner ist die objektorientierte Repräsentation eines Motion Problem Projects (siehe Kapitel 4.1).

[1] http://qt-project.org/

Dynamische Generierung

Die dynamische Beschreibung von TaskMapTypes (Kapitel 4.3) führt in der Implementierung des Models zu einer Besonderheit: Da die nötigen Informationen über das Aussehen einer speziellen TaskMap, insbesondere deren Parameter, erst zur Laufzeit zur Verfügung stehen, ist es nicht möglich, die Datenhaltung für eine TaskMap statisch zu erstellen, wie es oftmals üblich ist. Stattdessen wird sie bei Bedarf dynamisch, mit Hilfe des zugehörigen TaskMapTypes, nach einem Baukastensystem zusammengestellt.

Es wird zunächst eine Instanz der im TaskMapType angegebenen realisierenden Klasse erstellt und dieser ihre Identität (z.B. den realisierenden Typ) mitgeteilt. Anschließend wird für jeden Parameter, je nach Parametertyp, ein Objekt erzeugt und dieser wiederum konfiguriert: Er erhält Parametername, den angegebenen Standardwert und alle weiteren Informationen aus der Definition, die sein Verhalten beeinflussen können. Diese Objekte werden nun der TaskMap-Instanz hinzugefügt.

Sind diese Schritte erfolgreich gewesen, kann die generierte TaskMap dem Task hinzugefügt, und durch MotionDesigner verwendet werden.

Benachrichtigungssystem

Viele im Model enthaltenen Daten werden an mehreren Stellen im Programm gleichzeitig angezeigt oder verwendet und können durch unterschiedliche Ereignisse manipuliert werden. Werden Daten modifiziert, müssen lesend zugreifende Routinen entsprechend reagieren und z.B. die Anzeige aktualisieren. Offensichtlich muss also ein Weg gefunden werden, wie diese Änderungen bekannt werden.

Dafür bieten sich zwei Mechanismen an: zum einen eine aktive Überwachung des Models durch die lesenden Komponenten, zum anderen eine Benachrichtigung, sobald eine Änderung eintritt.

Ersteres bedeutet, dass jede Komponente das Model in regelmäßigen Abständen auf Änderungen hin überprüfen muss. Neben dem dafür nötigen Rechenaufwand kann es auch dazu kommen, dass Änderungen erst verhältnismäßig spät bemerkt werden oder zwischenzeitige Änderungen verloren gehen.

Ein Benachrichtigungssystem erzeugt ausschließlich dann Last, wenn tatsächlich eine Änderung stattfindet. Außerdem erfolgt die Benachrichtigung sofort und es können keine Ereignisse übergangen werden. Findet der Zugriff auf die Daten, entsprechend des Information Hidings, ausschließlich über entsprechende Methoden statt, kann sehr einfach in der schreibenden Methode ein entsprechendes Ereignis aufgerufen werden. Dadurch ist die Benachrichtigung für den Aufrufer der Methode absolut transparent.

Da das Model von der umgebenden Anwendung entkoppelt ist, können die einzelnen lesenden Komponenten nicht explizit informiert werden. Deshalb muss eine Schnittstelle entwickelt werden, über die beliebige Komponenten in das Benachrichtigungssystem eingebunden werden können. [GHJV95, S. 293] stellt dafür das Observer-Pattern vor, welches in einer

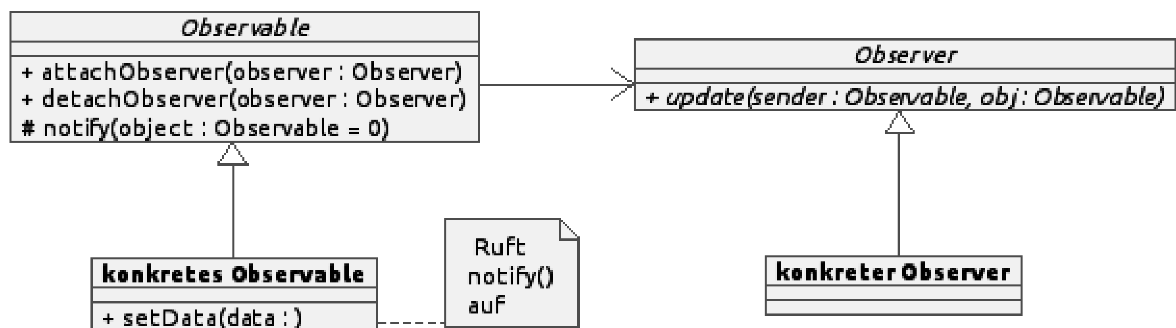


Abbildung 5.7: Im Model von MotionDesigner verwendete Umsetzung des Observer-Patterns

Variante, wie in Abbildung 5.7 gezeigt, verwendet wird. Hierbei können sich interessierte Objekte (sog. Observer) bei dem zu beobachtenden Objekt (Observable) registrieren. Bei einer Änderung wird bei allen registrierten Objekten eine bestimmte Methode aufgerufen. Der Mechanismus für die Benachrichtigung durch das Observable wird dabei in der Praxis in einer übergeordneten Klasse definiert, so dass die Anwendung sehr unkompliziert ist.

Das Observer-Pattern ist in der Implementierung sehr kompakt, sodass es kaum Einfluss auf die Größe der Codebasis des Models hat.

Eine Schwäche dieses Entwurfsmusters ist, dass es für gewöhnlich keine weiteren Informationen über die Änderung an den Observer übermittelt. Außerdem ist dem Observer die Quelle des Ereignisses nicht bekannt. Um zu ermöglichen, dass ein Observer mehrere Objekte gleichzeitig verfolgen kann - z.B. beobachtet eine TaskMap die Änderungen aller ihrer Parameter - und dabei noch nachvollziehen kann, welches Objekt sich geändert hat, wird in der genutzten Variante das aufrufende Objekt als Parameter mit übergeben.

Außerdem kann optional ein weiteres Observable-Objekt zu Referenzzwecken übergeben werden. Dies ermöglicht eine eingeschränkte Kaskadierung von Ereignissen. Das MotionProblemProject-Objekt, das alle Informationen über das aktuelle Projekt hält, leitet über diesen Weg alle Ereignisse seiner direkten Kinder weiter. Dadurch muss nur noch das MotionProblemProject überwacht werden, um über alle Änderungen im Model grob - da Details der Änderung verloren gehen - informiert zu werden.

Abhängigkeiten

Da das Model die in Kapitel 4 definierte Beschreibung eines Bewegungsproblems implementiert und den Adapter zwischen der Beschreibung und dem MotionProblem-Pakets von ORS darstellt, ist die Nutzung außerhalb von MotionDesigner naheliegend. Es kann zum Beispiel benutzt werden, um gespeicherte Probleme später auf einem Roboter auszuführen. Dafür ist es eminent wichtig, die Abhängigkeit von externen Bibliotheken - von der ORS-Bibliothek abgesehen - möglichst gering zu halten, um die begrenzten Ressourcen nicht unnötig zu belasten. Daher kommt speziell die Nutzung von Funktionalitäten aus dem Qt-Framework

nicht in Frage. Im Abschnitt „Trennung von Model und View“ wird auf die Konsequenzen für die Nutzung mit Qt näher eingegangen.

5.4.2 View

Wie bereits die TaskMap im Model, wird auch die Oberfläche zum Editieren der TaskMap-Parameter zur Laufzeit aus den Angaben des TaskMapTypes zusammengestellt. Dafür ist die Ansicht in unterschiedlichen Editor-Komponenten für einzelne Parameter aufgeteilt, die entsprechend der Spezifikation des genutzten TaskMapTypes zusammengesetzt sind.

Eine Komponente stellt den Zelleninhalt der Wert-Spalte in der Parametertabelle und, falls erforderlich, einen erweiterten Editor für den entsprechenden Parameter zur Verfügung. Dabei stehen unterschiedliche Komponenten als Module mit fester Schnittstelle zur Verfügung. Jedes Modul stellt dabei den Editor für bestimmte Parametertypen und unter Umständen einer spezifischen Ansicht. So gibt es zum Beispiel mehrere Module, die einen Editor für einen Parameter des Typs `string` anbieten: Ein einfaches Textfeld mit freier Texteingabe ist das Standardmodul. Das Modul „BodySelect“ ermöglicht keine freie Eingabe, sondern bietet dem Benutzer eine Liste aller namentlich definierten Bodies im ORS-Model zur Auswahl an und speichert die Auswahl als String im Model.

In der Definition des TaskMapTypes kann über das Element `ui` angegeben werden, ob ein vom Standard abweichendes Modul verwendet werden soll.

Dieses Verfahren hat den Vorteil, dass der Entwickler direkt in der TaskMapType-Definition angeben kann, welche Art Eigenschaft der Parameter repräsentiert, ohne dass jede Variation über den Parametertypen abgebildet werden muss. Denn im Parameter-Model ist es belanglos, ob der gespeicherte String nun der Name eines Body, eine Transformation oder ein unbedeutender Kommentar ist, syntaktisch ist es in jedem Fall lediglich eine Zeichenkette und eine Unterscheidung würde zu großen Redundanzen im Code führen. Eine Validierung des Inhalts kann auf Seiten des Models in der TaskMap-Klasse erfolgen, die auch den Zusammenhang, in dem der Parameter steht, kennt, sowie auf der Seite der View, bereits bevor eine Benutzereingabe an das Model gesendet wird.

5.4.3 Trennung von Model und View

Qt besitzt für die Kommunikation zwischen Objekten ein System aus sendenden „Signals“ und empfangenden „Slots“. Dieses System soll Probleme der in älteren Toolkits oft genutzten Callbackfunktionen lösen, die unter anderem mit ihrer fehlenden Typensicherheit leicht zu schweren Fehlern führen können ([Dig13b]).

Ein Beispiel für die Nutzung des Systems ist ein Speichern-Button: Der Benutzer löst den Button durch einen Mausklick aus. Das Button-Objekt emmitiert daraufhin das Signal „ausgelöst“, das von einem verbundenen Slot des geöffneten Projekts empfangen wird und seine Daten daraufhin speichert.

Klassen, die dieses System verwenden, können aber nicht mehr ohne das Qt-Framework arbeiten. Jede Klasse, deren Objekte Signale absetzen oder empfangen sollen, muss direkt oder indirekt von der Qt-Basisklasse `QObject` abgeleitet sein. Außerdem ist der Quelltext der Klassen kein reines C++ mehr. Es ist daher notwendig, die Dateien erst mit dem dem sog. meta-object compiler (moc) zu behandeln, der aus den darin enthaltenen Informationen über Signale und Slots C-Code generiert. Nachfolgend muss die Klasse durch einen C++-Preprozessor so umgewandelt werden, dass sie Standard C++ entspricht.

In Anbetracht der in Kapitel 5.4.1 geforderten Unabhängigkeit des Models von externen Bibliotheken ist offensichtlich, dass dieses System, obleich es viele Vorteile bietet, nicht für die Kommunikation zwischen Model und View geeignet ist. Da die Nutzung von Qt ohne Signale und Slots nahezu unmöglich ist, insbesondere bei teilweise dynamisch generierten Oberflächen wie in MotionDesigner, bilden unterschiedliche Maßnahmen einen Adapter zwischen den Bestandteilen.

1. Zentrale Klassen des Models wurden abgeleitet und um Qt-Funktionalitäten erweitert.
   a) Die Klasse `MotionProjectProblem`, die hierarchisch im Model zu oberst angeordnet ist, sendet bei Änderungen in ihrem Datenbestand (z.B. Laden oder Erstellen eines neuen Projekts, Änderung von Einstellungen, etc.) Signale aus. Des weiteren beobachtet sie alle direkt untergeordneten Objekte als Observer und wandelt darüber empfangene Ereignisse in Signale um.
   b) Die zwei Klassen, die eine Liste von TaskMapTypes oder TaskMap verwalten, wurden direkt um die Funktionen eines `QAbstractListModels` erweitert, sodass sie direkt als Qt-Model für Darstellungen in Listenform geeignet sind.
2. Widgets, also Objekte, die Teile der grafischen Oberfläche darstellen, sind um das Observer-Pattern erweitert, um in der Model-Hierarchie tiefergelegene Objekte direkt zu beobachten und auf View-Ebene entweder in Signale umzuwandeln oder entsprechende Operationen durchzuführen.

Die Maßnahme 2 scheint zunächst suboptimal, da es die Kopplung von View und Model deutlich erhöht. Nicht nur nutzt es den, bisher nur im Model zu findenden, Observer, es sind auch weitreichende Kenntnisse über die Implementierung des Models nötig. Eine geringe Kopplung zwischen Modulen ist nach [LL10, S. 413] aber ein wesentliches Entwurfsziel im objektorientierten Programmieren.

Insbesondere die Methode 1b trennt die einzelnen Aspekte wesentlich sauberer: Sie kapselt das Wissen über die Verbindung des MotionDesigner-Models und eines Qt-Models in einer eigenen Klasse. Je nachdem, ob das Model für die Verwendung mit oder ohne Qt gewünscht ist, wird die Schnittstelle durch Ableiten der Modelklasse entsprechend gewählt

Durch die dynamische Generierung einzelner Komponenten des Models ist das Ableiten aller Model-Klassen aber schwierig: Es muss, je nachdem ob Qt vorhanden ist oder nicht, eine Instanz der richtigen Klasse erstellt werden. Eine einfache Weiche kommt hierbei nicht

in Frage, denn zum einen ist dem Objekt nicht bekannt, ob es die Instanz einer Qt-Adapter-Klasse oder der einfachen Model-Klasse ist, zum anderen existieren die Qt-Adapter-Klassen unter Umständen gar nicht, wodurch das Kompilieren fehlschlagen würde.

6 Zusammenfassung und Ausblick

Die einheitliche Formulierung der einzelnen Aufgaben des Problems als Kostenterme ermöglicht das Erstellen einer sehr klar strukturierten Anwendung, da alle Aufgaben auf dem gleichen Weg angelegt werden können, wie der in dieser Arbeit entwickelte MotionDesigner zeigt.

Insbesondere für das Abgleichen der Kosten-Präzisionen aufeinander stellen sich die Visualisierungen und die Simulation des Ergebnisses dabei in der Benutzung als sehr hilfreich heraus, denn wenn alle nötigen Regeln identifiziert sind, ist die richtige Priorisierung der einzelnen Aufgaben maßgeblich dafür, dass der Optimierer eine „schöne“ Bewegung finden kann.

Das schrittweise Hinzufügen von Tasks erleichtert das Erstellen eines Problems immens. Es werden zunächst offensichtliche Tasks wie das Erreichen des Zielpunktes angelegt und dann, unter Betrachtung des bisherigen Ergebnisses, immer weitere Details eingebracht, bis die Bewegung den Erwartungen entspricht. Dadurch entfällt für den Anwender die Aufgabe, eine Bewegung komplett in allen ihren Bestandteilen zu beschreiben, sondern er betrachtet eine Bewegung und schlägt eine einzelne Verbesserung vor.

Das in Kapitel 4 vorgestellte Datenformat konnte bereits während der Entwicklung von MotionDesigner zeigen, dass es sehr gut für den Einsatz geeignet ist. Änderungen am Format, wie etwa das Hinzufügen der Kostenfunktion zur Task-Beschreibung, konnten schnell eingeführt werden und auch manuelle Änderungen direkt in den Dateien waren ohne Probleme durchführbar. Leider besitzt der Parser des KeyValueGraphs nur sehr begrenzte Möglichkeiten zur syntaktischen Validierung und kann keine semantische Validierung durchführen, sodass die Implementierung sich für ein klar definiertes Datenformat, wie hier vorhanden, weniger eignet. Die Stärke liegt eindeutig in der Repräsentation von komplett freien Datenstrukturen. Das Ersetzen des Parsers, z.B. durch im Bereich des Compilerbaus übliche Verfahren wie regulären Ausdrücken, kann sicher die Stabilität und Fehlertoleranz erhöhen, schränkt aber auch die Flexibilität ein.

Es zeigte sich des Weiteren, dass das ORS-Paket nicht direkt auf die Anwendung in graphischen Oberflächen mit hoher Parallelität ausgelegt ist. So ist es zum Beispiel nötig, den ORS-Graphen für jede Benutzung zu kopieren, da in den Objekten Informationen über den aktuellen Zustand gespeichert sind. Benutzen beispielsweise Optimierer und Animation den gleichen ORS-Graphen, kann es dazu kommen, dass die Animation während des Optimiervorgangs von der gewünschten Bewegung abweicht oder das der Optimierer die Trajektorie nicht von der gewünschten Startkonfiguration aus erstellt. Da mit der Kopie des ORS-Graphs auch stehts alle Bodies, Shapes, Joints, etc. kopiert werden, kommt es zu großen Redundanzen, die den Speicherverbrauch stark steigern.

6.1 Ausblick

Eine interessante Erweiterung des ORS-Graphen wäre, die Beschreibung des Roboters von der Beschreibung der Umwelt zu trennen. So könnte der Anwender durch das Austauschen der Umgebung testen, wie sich das erstellte Problem unter geänderten Bedingungen verhält. Da oftmals auch viele Projekte für den gleichen Roboter gedacht sind, kann so ebenfalls Redundanz in der Datenhaltung verringert werden. Neben dem reduzierten Speicherverbrauch erlaubt das insbesondere, Änderungen am Robotermodell an einer zentralen Stelle ausführen zu können statt für jedes einzelne Projekt.

Für die Entwicklung von Bewegungen ist außerdem die Möglichkeit, Gruppen von Tasks unabhängig vom Projekt separat zu speichern, als sehr nützlich vorstellbar. Dadurch könnten oft benötigte Tasks, wie zum Beispiel die Grenzen der Gelenkauslenkung oder maximal erlaubte Kräfte, in Vorlagen ausgelagert werden, die dann direkt zum aktuellen Projekt hinzugefügt werden könnten.

Literaturverzeichnis

[BSS07] E. Brommundt, G. Sachs, D. Sachau. *Technische Mechanik: eine Einführung*. Oldenbourg, 2007. (Zitiert auf Seite 11)

[Chi13] S. Chitta. MoveIt!, 2013. URL http://www.willowgarage.com/blog/2013/05/06/moveit. (Zitiert auf Seite 17)

[Dig13a] Digia. Model/View Programming, 2013. URL http://qt-project.org/doc/qt-5.0/qtwidgets/model-view-programming.html. (Zitiert auf Seite 40)

[Dig13b] Digia. Signals & Slots, 2013. URL http://qt-project.org/doc/qt-5.0/qtcore/signalsandslots.html. (Zitiert auf Seite 43)

[DMV12] DMV. Nevada DMV Issues First Autonomous Vehicle Testing License to Google, 2012. URL http://www.dmvnv.com/news/12005-autonomous-vehicle-licensed.htm. (Zitiert auf Seite 7)

[Gar13] W. Garage. Willow Garage History, 2013. URL http://www.willowgarage.com/pages/about-us/history. (Zitiert auf Seite 17)

[GHJV95] E. Gamma, R. Helm, R. Johnson, J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995. (Zitiert auf den Seiten 40 und 41)

[Hla10a] V. Hlavac. Motion planning methods, 2010. URL http://cmp.felk.cvut.cz/~hlavac/TeachPresEn/55IntelligentRobotics/100MotionPlanningMethods.pdf. (Zitiert auf den Seiten 5 und 16)

[Hla10b] V. Hlavac. Path Planning In Robotics, 2010. URL http://cmp.felk.cvut.cz/~hlavac/TeachPresEn/55IntelligentRobotics/090PathPlanningInRobotics.pdf. (Zitiert auf den Seiten 5 und 14)

[IAT13] IAT. Assistenzroboter FRIEND, 2013. URL http://www.iat.uni-bremen.de/sixcms/detail.php?id=1090. (Zitiert auf Seite 7)

[Jaz10] R. Jazar. *Theory of Applied Robotics: Kinematics, Dynamics, and Control*. Springer, 2010. (Zitiert auf Seite 12)

[Kni13] W. Knight. Kollege Roboter am Fließband. *Technology Review*, 2013. URL http://heise.de/-1972126. (Zitiert auf Seite 7)

[Knu08] D. Knuth. *The Art of Computer Programming: Fundamental algorithms*. The Art of Computer Programming. Addison-Wesley, 2008. (Zitiert auf Seite 39)

[L+13] M. Likhachev, et al. Search-Based Planing Lab, 2013. URL http://www.sbpl.net. (Zitiert auf Seite 16)

[LL10] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt-Verlag, 2010. (Zitiert auf den Seiten 40 und 44)

[ROS13] ROS. Introduction, 2013. URL http://wiki.ros.org/ROS/Introduction. (Zitiert auf Seite 17)

[SC13] I. A. Sucan, S. Chitta. MoveIt!, 2013. URL http://movit.ros.org. (Zitiert auf den Seiten 5 und 17)

[ŞMK12] I. A. Şucan, M. Moll, L. E. Kavraki. The Open Motion Planning Library. *IEEE Robotics & Automation Magazine*, 19(4):72–82, 2012. doi:10.1109/MRA.2012.2205651. http://ompl.kavrakilab.org. (Zitiert auf Seite 18)

[Tou13a] M. Toussaint. Introduction to Robotics: Kinematics, 2013. URL http://ipvs.informatik.uni-stuttgart.de/mlr/marc/teaching/13-Robotics/02-kinematics.pdf. (Zitiert auf den Seiten 5 und 11)

[Tou13b] M. Toussaint. Motion Generation, 2013. Internal paper. (Zitiert auf den Seiten 18 und 28)

[Vol13] Volvo. Volvo Car Group startet weltweit einzigartiges Pilotprojekt zum autonomen Fahren auf öffentlichen Straßen, 2013. URL http://www.volvocars.com/de/top/about/news_events/pages/press.aspx?itemid=428. (Zitiert auf Seite 7)

[ZRD+13] M. Zucker, N. Ratliff, A. Dragan, M. Pivtoraiko, M. Klingensmith, C. Dellin, J. A. D. Bagnell, S. Srinivasa. CHOMP: Covariant Hamiltonian Optimization for Motion Planning. *International Journal of Robotics Research*, 2013. (Zitiert auf den Seiten 15 und 16)

Alle URLs wurden zuletzt am 18. 11. 2013 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift