Studienarbeit

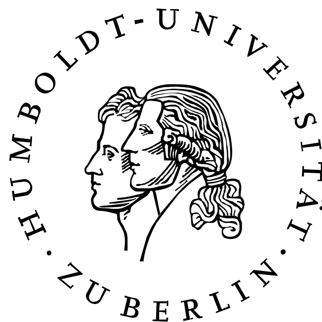
Thema: Integrierte Entwicklungsumgebung zur Bewegungssteuerung humanoider Roboter

Autor: Christian Thiele

Betreuer: Dr. Manfred Hild

Lehrstuhl für Künstliche Intelligenz
Prof. Dr. Hans-Dieter Burkhard

Oktober 2007



Inhaltsverzeichnis

1 Einleitung **1**

2 Steuerung humanoider Roboter **2**

2.1 Robotersteuerungen 2

2.2 Steuerung mit Keyframes 4

2.3 Keyframe-Netze 6

2.4 Manipulation mit neuronalen Netzen 7

2.5 Neuronale Steuerung 8

3 Der Motion Editor **9**

3.1 Grundidee der Software des Humanoid Team Humboldt 9

3.2 Die Software 10

3.3 Evolutionäre Entwicklung 17

4 Hardware-Plattform **19**

4.1 Die Roboter des Humanoid Team Humboldt 19

4.2 Steuerungsplatinen der Roboter 20

4.3 Autonomer Betrieb der Roboter 21

4.4 Kommunikation zwischen Entwicklungsumgebung und Roboter 22

5 Zusammenfassung und Ausblick **24**

Anhang 1: Neuronaler Bytecode .... **i**

Literatur .... **xv**

Abstract. Die vorliegende Arbeit beschäftigt sich mit der Bewegungssteuerung der humanoiden Roboter des Humanoid Team Humboldt (HTH). Vorgestellt wird neben den theoretischen Ansätzen, die der Steuerung der Roboter des HTH zugrunde liegen vor allem die im Rahmen dieser Studienarbeit entwickelte PC-Software zur Erstellung von Bewegungen und die Übertragung der so erstellten Bewegungen auf autonome Hardware.

1 Einleitung

Die Bewegungssteuerung ist bei komplexen Maschinen wie humanoiden Robotern in mehrerer Hinsicht schwierig. Zum einen gilt es, eine hohe Anzahl an Freiheitsgraden anzusteuern, zum anderen muss die Steuerung auf die Umwelt (in Form von Sensorsignalen) reagieren und den Körper beispielsweise stabilisieren. Ein weiteres Problem stellt die Tatsache dar, dass autonome Roboter mit wenig zur Verfügung stehender Rechenleistung gesteuert werden müssen.

Diese Studienarbeit stellt die Bewegungssteuerung des Humanoid Team Humboldt (HTH) an der Arbeitsgruppe Künstliche Intelligenz der Humboldt-Universität zu Berlin vor. Gesteuert werden kleine autonome humanoide Roboter, die 19 Aktuatoren zur Bewegung des Körpers besitzen. Dabei handelt es sich um einfache Getriebemotoren, diese werden in dieser Arbeit auch als Gelenke bezeichnet.

Zunächst beleuchte ich dabei die Bewegungssteuerung im Allgemeinen und gehe auf die von uns zunächst gewählte Keyframe-Methode im Speziellen ein. Darauf folgend stelle ich im längsten Kapitel, dem Kapitel 3, die von mir im Rahmen dieser Studienarbeit entwickelte PC-Software *Motion Editor* vor, mit der das Team die Bewegungen der Roboter erstellt. Dabei wird die Software und deren Bedienung ebenso vorgestellt, wie deren Entwicklungsgeschichte nachgezeichnet wird.

Im darauffolgenden Kapitel wird über die Hardware und die Kommunikation zwischen dem Motion Editor und der Hardware berichtet. Ein besonderes Augenmerk liegt auch hier auf der auf dem Roboter laufenden Software und die dabei von mir entwickelten Teile zur Bewegungssteuerung. Im Anhang wird der dabei verwendete und im Rahmen dieser Arbeit entwickelte neuronale Bytecode mit dem dazugehörigen Assembler detailliert vorgestellt.

Diese Arbeit soll zeigen, dass eine Keyframe-Steuerung mit anschließender neuronaler Korrektur als Zwischenschritt zur vollständig neuronalen Steuerung gut geeignet ist und auf diese Weise schnell respektable Ergebnisse erzielt werden können. Roboter mit der hier vorgestellten Bewegungssteuerung konnten bei den RoboCup German Open 2007, den deutschen Meisterschaften im Roboter-Fußball, den dritten Platz belegen.

2 Steuerung humanoider Roboter

2.1 Robotersteuerungen

Seit es erste einfache Roboter gibt, existiert auch das Problem, diese zu steuern. Dabei bezieht sich die Steuerung zunächst auf den gesamten Prozess von der Wahrnehmung bis hin zur Ansteuerung der Aktuatoren. Dieser Gesamtprozess wird jedoch je nach Ziel und Komplexität in Unteraufgaben geteilt: Häufig werden zunächst die Sensordaten der Umwelt verarbeitet, aus den gewonnenen Informationen ein gewünschter Bewegungspfad berechnet und anschließend die für diesen Pfad nötige Ansteuerung der Aktuatoren berechnet.

Bei den einfachsten Robotern ist solch eine Trennung oft nicht nötig, Braitenberg hat gezeigt, dass einfachste neuronale Steuerungen für fahrende Roboter mit zwei Freiheitsgraden und zwei Sensoren reichen, um einfaches ausweichendes Verhalten zu produzieren [Bra84]. An dieser Stelle ist keine komplexe Weltmodellierung und Pfadberechnung nötig.

Im Folgenden soll es um den eigentlichen Ansteuerungsprozess gehen: Wie steuere ich Motoren an, wenn ein vorgeschaltetes Verhalten eine bestimmte Bewegung – beispielsweise „Linksherum fahren" – fordert? Bei einfachen Robotern, wie Braitenberg sie beschrieben hat, ist dieses Problem leicht zu lösen – für Roboter, die kompliziertere Bewegungen mit vielen Freiheitsgraden absolvieren sollen, ist diese Fragestellung schwieriger.

Die einfachste Idee ist, die Trajektorien[1] jedes einzelnen Aktuators als Funktionen über die Zeit zu speichern, wobei meist die Positionen zu diskreten Zeitpunkten (beispielsweise 100 Mal pro Sekunde) gespeichert und später angesteuert werden [BS79]. Die Diskretisierung ist in modernen digitalen Systemen unumgänglich. Solch ein Verfahren wurde schon Ende der 1960er Jahre benutzt, um an grafischen Arbeitsstationen Elemente zu animieren, indem man die Bewegungskurven grafisch eingab [Bae69], zur Robotersteuerung ist die zu speichernde Datenmenge jedoch zu groß.

---

[1] Als *Trajektorie* bezeichnet man die Ortsraumkurve, also im einfachsten Fall die Motorpositionen abhängig von der Zeit.

Eine Reduktion der Datenmenge kann man erzielen, indem man nicht mehr für jeden Zeitschritt die anzusteuernden Werte speichert, sondern nur für einige wenige ausgewählte Zeitpunkte. Die dann fehlenden Werte für nicht gespeicherte Zeitschritte interpoliert man. Auf diese Weise wird Rechenleistung zugunsten des Speicherplatzes investiert. Zeitpunkte, zu denen gespeicherte Werte vorliegen, werden als „Keyframes" und die Technik als „Keyframe-Technik" bezeichnet. Auf diese Technik werde ich im nächsten Kapitel detaillierter eingehen.

Keyframe-Techniken eignen sich besonders zur wiederholgenauen Ansteuerung von Industrieanlagen und ähnlichen Robotern. Die Einbindung von Sensorinformationen über die Außenwelt oder den inneren Zustand des Roboters ist bei solchen Systemen auf Ansteuerungsebene fast immer unnötig. Anders sieht dies bei mobilen Robotern aus. Diese müssen Änderungen der Umwelt ausgleichen, beispielsweise unterschiedliche Bodenbeschaffenheiten oder Gefälle. Besonders wichtig ist die Einbindung von Sensorinformationen bei humanoiden Robotern, da diese durch die im Verhältnis zur Höhe des Schwerpunkts geringe Auflagefläche recht instabil sind.

Ein Ansatz dafür ist die nachträgliche Korrektur der durch Keyframes erhaltenen Ansteuerungsdaten. Als besonders geeignet haben sich hierfür künstliche neuronale Netze erwiesen. Ein weiterer Ansatz sind komplett neuronale Steuerungen, wobei diese aufgrund ihrer Komplexität meist schwer manuell zu entwickeln sind. In [Hil07] wird daher ein Verfahren vorgestellt, Keyframes mit sogenannten neuronalen Monoflops zu ersetzen und somit einen Übergang zwischen Keyframe-Ansteuerung und rein neuronaler Ansteuerung zu erreichen.

Zum Finden komplexer neuronaler Steuerungsnetze wird oft die Methode der künstlichen Evolution eingesetzt (zur künstlichen Evolution siehe z. B. [NF00]). Häufig wird dabei nicht direkt auf Hardware evolviert, sondern zunächst in Simulatoren [Hei07].

Bei Keyframe-Techniken werden fast immer einfach die anzusteuernden Motordaten an den Keyframes gespeichert. Zum Teil wird hierbei für humanoide Roboter jedoch abgewichen. Bei [YI06] wird ein System vorgestellt, bei dem die Bewegungen des Unterkörpers nur durch Parameter dargestellt werden, die festlegen, wo sich der Fuß in Abhängigkeit zur Hüfte befinden soll. Die genauen Motorpositionen von Hüfte, Kniegelenk und Fußgelenken werden dann so berechnet und interpoliert, dass der Masseschwerpunkt des Roboters nie die konvexe Hülle seiner Berührungspunkte mit dem Boden verlässt und dieser daher stabil ist. Auf echter Hardware wurden jedoch schon bei leichten Seitbewegungen mit damit verbundenem Abheben des Fußes Instabilitäten beobachtet. Der Kick eines Balles (und das damit verbundene Stehen auf einem Fuß) wurde jedoch nur in einem Simulator durchgeführt. Die geringe Fußflä-

che bei gleichzeitig hohem Getriebespiel, das sich über die kinetische Kette vom Fuß bis zum Oberkörper fortsetzt, lässt diese Methode wenig tauglich für den Einsatz echter Roboter erscheinen.

Eine weitere Möglichkeit zur Bewegungsteuerung, die jedoch im praktischen Einsatz ausschließlich mit evolutionären Ergebnissen arbeitet, ist das genetische Programmieren [BZA02]. Dabei wird der Roboter ausschließlich von einer Folge von einfachen Befehlen gesteuert, die Programme hierfür werden jedoch mithilfe von evolutionären Algorithmen gefunden. Als Befehle stehen einfache Rechenbefehle wie beispielsweise ADD und MUL ebenso zur Verfügung wie Befehle zum Einlesen von Sensorinformationen (z.B. SENSE) und zum Schreiben von Motordaten (z.B. MOVE). Dem Lehrstuhl für Systemanalyse an der Universität Dortmund war es so möglich, Laufbewegungen für simulierte Roboter verschiedener Morphologien[2] zu erstellen.

2.2 Steuerung mit Keyframes

Wie beschrieben stellt eine Keyframe-basierte Steuerung eine der einfachsten Steuerungen dar. Zu festgelegten Zeiten oder in festen Intervallen wird dabei eine komplette Roboterpose gespeichert und die zugehörigen Ansteuerungswerte für Zeitschritte zwischen Keyframes interpoliert. Die Übergänge zwischen Keyframes werden dabei *Transitionen* genannt.

Bereits 1979 haben Badler und Smoliar in ihrem Paper „Digital Representations of Human Movement" [BS79] aufgezeigt, dass komplexe menschliche Bewegungen auf diese Weise gespeichert werden können.

Der eigentliche Aufwand dieser Methode liegt in der Interpolation. Die einfachste Möglichkeit ist dabei die lineare Interpolation. Auf hochpräzisen Systemen ist dies oft nicht ausreichend, da die Unstetigkeiten an den Keyframes zu Problemen führen können. In diesem Fall werden höherpolynomielle Funktionen zur Interpolation benutzt. Hochpolynomielle Funktionen, die über mehrere Punkte interpolieren, neigen jedoch zu Oszillationen (als „Runges Phänomen" bekannt), weshalb stückweise polynomiell interpoliert wird und somit niedrigere Polynome genügen. Dabei wird nicht über eine längere Zeit eine Interpolation vorgegeben sondern für jeden einzelnen Abschnitt zwi-

[2] Die *Morphologie* ist in der Biologie die Lehre von der Struktur und Form von Lebewesen. In der Robotik wird der Begriff auf die Gestalt von Robotern übertragen.

schen zwei Keyframes. Verwendet werden dabei sogenannte Spline-Funktionen, die den Vorteil haben, auch in der zweiten Ableitung noch stetig zu sein [SB85].

Die häufigste Anwendungsform sind kubische Splines, da diese bei geringem Rechenaufwand gute Ergebnisse erzielen [Tha89] [Mem03]. In manchen Systemen, wie dem in [KB03] entwickelten *Motion Creating System* für den Unterhaltungsroboter Qrio, kann dynamisch eine von mehreren Interpolationsmethoden gewählt werden.

Auf unseren Robotern der Serie A (HTH/A) wird auf höherpolynomielle Interpolation verzichtet und linear interpoliert, da die hochfrequenten Störungen an den Keyframes durch die Trägheit der Aktuatoren ausgeglichen werden. Sollte die lineare Interpolation an manchen Stellen nicht ausreichen, kann die Kurvenform mit zusätzlichen Keyframes korrigiert werden.

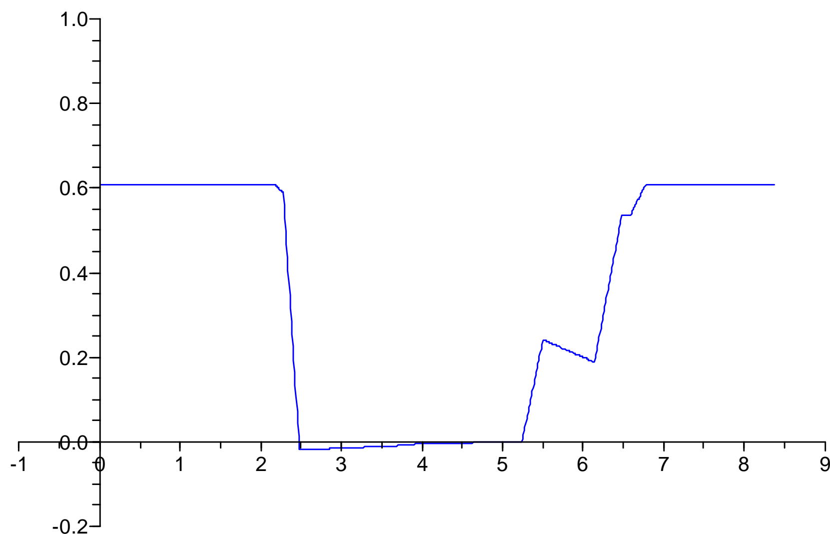


Abbildung 1: Echte, angesteuerte Trajektorie eines Motors auf Keyframe-Basis (auf der x-Achse ist die Zeit abgetragen, auf der y-Achse die Motorposition). Man sieht, dass über lange Zeiten lineare Abschnitte vorhanden sind.

2.3 Keyframe-Netze

Die meisten Keyframe-basierten Programme zur Erstellung von Bewegungen erlauben nur Transitionen zwischen genau zwei aufeinanderfolgenden Keyframes, so beispielsweise der mit dem unseren Robotern zugrunde liegende Roboterbausatz *Bioloid* ausgelieferte *Motion Editor* und das ebenfalls *Motion Editor* genannte Programm, das in [WH04] zur Erstellung von Tai-Chi-Bewegungen für einen humanoiden Roboter verwendet wird. Schleifen und der Wechsel zwischen verschiedenen Bewegungen obliegt so höheren Steuerungsebenen.

Unsere Bewegungssteuerung soll diese Aufgabe bereits auf niedriger Ebene übernehmen. So wurde es möglich gemacht, beliebige Keyframes mit Transitionen zu verbinden und so Schleifen zu bilden. Das Verlassen der Schleifen oder das Ändern des Verhaltens (beispielsweise Aufstehen statt Laufen, da der Roboter umgefallen ist) wird über Keyframes realisiert, von denen mehrere, selektive Transitionen wegführen. Zu diesem Zweck können Transitionen mit sogenannten Selektoren versehen werden. Im Bewegungsnetz ist dabei immer ein Selektor global aktiv. Gibt es abgehend von einem Keyframe mehrere Transitionen wird zunächst gesucht, ob es eine solche gibt, deren Selektor dem derzeitig im Bewegungsnetz gesetzten entspricht. Wird diese gefunden, wird diese Transition genommen, ansonsten die Standardtransition (eine Transition ohne Selektor).

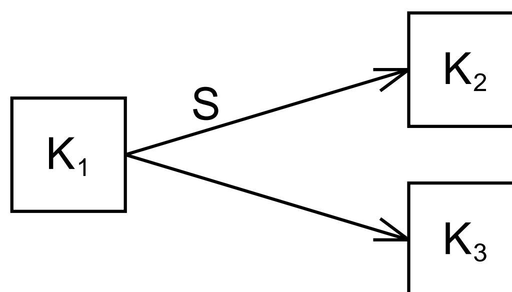


Abbildung 2: Vom Keyframe 1 gibt es zwei abgehende Transitionen. Ist der globale Selektor auf „S“ gesetzt, wird die Transition zu Keyframe 2 genommen in allen anderen Fällen jene zu Keyframe 3.

Auf diese Weise ist es möglich, in einem einzelnen Keyframe-Netz verschiedenste Bewegungsabläufe zu speichern und über das einfache Setzen eines Selektors die gewünschte Bewegung aufzurufen.

2.4 Manipulation mit neuronalen Netzen

Ohne eine Stabilisierung durch Sensordaten ist es bei instabilen Systemen wie humanoiden Robotern schwierig, Bewegungen nur mit Keyframes zu erstellen. In „Realization of Tai-Chi Motion Using a Humanoid Robot“ [WH04] wird auf die mühsame Anpassung auf echte Hardware eingegangen und dieser Schritt als wichtigster und zeitintensivster Prozess bezeichnet.

Auch in unserem Team hat sich gezeigt, dass die manuelle Anpassung von Bewegungen für verschiedene Roboter zeitintensiv ist. Wünschenswert ist es, mit neuronalen Strukturen einzelne Ansteuerungswerte abhängig von Sensorinformationen zu manipulieren.

Entwickelt wurde eine einfache Sprache zur Beschreibung neuronaler Netze, die auf sämtliche Sensorwerte (Beschleunigungsdaten und Motorpositionen) und auf die vorher per Keyframe-Verfahren berechneten Ansteuerungsdaten zugreifen kann und neue Ansteuerungsdaten schreiben kann. Dabei ist es möglich, mehrere so erstellter Programme auf dem Roboter abzulegen und pro Transition ein anderes Programm zu wählen, um bestimmte Stabilisierungen (beispielsweise fürs Laufen) für andere Prozesse (beispielsweise beim Aufstehen) abzuschalten.

Der in Kooperation mit der Universität Osnabrück im Rahmen dieser Studienarbeit entwickelte Code ist ein einfacher Bytecode mit zugehöriger Assemblersprache, der wenige Befehle zum Setzen von Gewichten, zum Lesen von Werten, zum Multiplizieren und Addieren dieser, für eine Aktivierungsfunktion und zum Schreiben des Ergebnisses zur Verfügung stellt. Das folgende Beispiel simuliert ein Neuron, das als ersten Eingang einen anzusteuernden Motorwert hat (Gewicht 1,0; Motorwert an SpinalCord-Position SC_{31}), der mit einem zweiten Eingang (Gewicht 0,6; Beschleunigungswert an SpinalCord-Position SC_3) ausgeglichen wird.

```
weight 1
read 31
mac
weight 0.6
read 3
mac
tanh
res 31
```

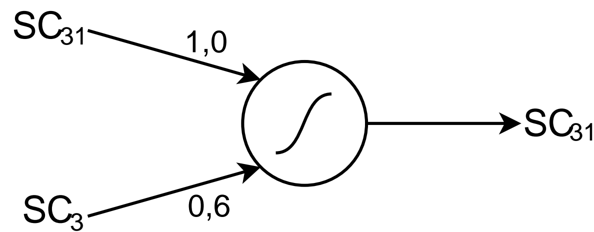


Abbildung 3: Grafische Darstellung des Neurons aus dem vorherigen Beispielprogramm

Zusätzlich zu Ansteuerungs- und Sensorwerten stehen freie Neuronenspeicher zur Verfügung, womit größere Netze erstellt werden können. Eine komplette Darstellung der Sprache findet sich in Anhang 1.

2.5 Neuronale Steuerung

Eine vollständig neuronale Bewegungssteuerung eines humanoiden Roboters ist sehr kompliziert umzusetzen. Verwendet werden kann ein einfacher Oszillator, der in der Lauffrequenz schwingt und über dessen Ausgänge die entsprechenden Motoren für Beine und Arme gesteuert werden. In unserem Team wurden solche Steuerungen in Handarbeit entwickelt, die auf unseren Robotern gute Laufergebnisse erzielen. Meist jedoch sind vollständig neuronale Steuerungen das Ergebnis einer künstlichen Evolution oder werden mit der bereits erwähnten Methode aus [Hil07] aus Keyframe-Netzen gewonnen und anschließend optimiert und erweitert.

Die Struktur der im vorherigen Unterkapitel vorgestellten, sehr einfachen Assembler-Sprache erlaubt es, auch ausschließlich neuronale Steuerungen zu implementieren, indem vorher per Keyframe-Steuerung berechnete Werte einfach nicht als Eingangswerte für Neuronen verwendet werden, sondern nur mit Sensor-Inputs gearbeitet wird.

3 Der Motion Editor

3.1 Grundidee der Software des Humanoid Team Humboldt

Der Bau der Roboter des Humanoid Team Humboldt begann im Januar 2006. Zur Bewegungssteuerung wurde zunächst der im zugrundeliegenden Bioloid-Bausatz mitgelieferte CM5-Prozessor verwendet, die Bewegungen selbst wurden mit der mitgelieferten Windows-Software namens *Motion Editor* erstellt. Schon vorher war klar, dass diese Steuerung unseren Anforderungen nicht genügen wird und wir komplexe Keyframe-Netze mit der Möglichkeit zur Einkopplung von Sensorinformationen benötigen würden. Daher waren acht, miteinander verbundene, kleine Steuerungsplatinen am Roboter vorgesehen, die jeweils bis zu drei Motoren steuern sollten (diese Platinen werden im Folgenden *AccelBoards* genannt). Nach den ersten Erfahrungen mit dem Motion Editor des Bioloid-Bausatzes und einer Analyse der benötigten Funktionalität begann ich Ende März 2006 mit der Entwicklung des *HTH Motion Editors* (im Folgenden ist mit Motion Editor immer – sofern nicht explizit anders erwähnt – der HTH Motion Editor gemeint).

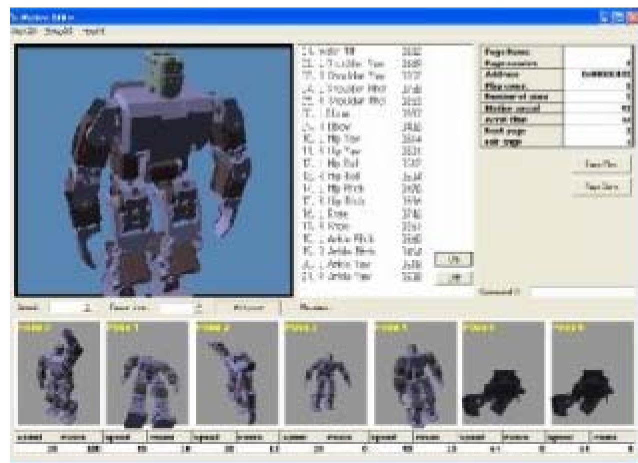


Abbildung 4: Der Bioloid Motion Editor. Oben links wird der Roboter grafisch dargestellt, rechts daneben können die Motoren eingestellt werden. Im unteren Bereich werden die einzelnen Posen (die Keyframes) einer Bewegung dargestellt.

Ausgangspunkt war zunächst die einfache Keyframe-Steuerung, da die Erfahrung mit dem Motion Editor des Bioloid-Bausatzes zeigte, dass so relativ schnell Bewegungen erstellt werden können. Erweitert war jedoch vorgesehen, Keyframe-Netze einzusetzen, die Software sollte also vor allem die Keyframe-Zusammenhänge grafisch darstellen können. Auf eine grafische dreidimensionale Darstellung der Roboterposen hingegen konnte verzichtet werden, da immer direkt am Roboter gearbeitet werden sollte. Dafür war es nötig, vom Motion Editor aus den Roboter mit einer Pose ansteuern und Bewegungen direkt abspielen zu können. Weiterhin sollte es möglich sein, einzelne Gelenke oder den ganzen Roboter zu entspannen, an diesem dann neue Posen zu „kneten" und anschließend die Motorpositionen auszulesen. Damit die AccelBoards die Bewegungssteuerung übernehmen können, wenn der PC nicht angeschlossen ist, sollte es per Knopfdruck möglich sein, die Bewegungsdaten auf die AccelBoards zu überspielen.

Eine andere Erweiterung im Vergleich zum Motion Editor des Bioloid-Bausatzes stellt die Möglichkeit dar, das Drehmoment, mit dem die einzelnen Motoren angesteuert werden, ebenfalls einzustellen. Hier wurde ein ähnliches lineares Interpolationsverfahren wie für die Positionen genutzt, jedoch ist es im Gegensatz zu diesem möglich, an den Keyframes Sprünge einzufügen, sodass direkt von einem Zeitschritt zum anderen das nötige Drehmoment angesteuert werden kann.

3.2 Die Software

Praktisch sämtliche Funktionalität des Motion Editors wird über ein Hauptfenster, wie in Abbildung 5 auf der nächsten Seite dargestellt, zur Verfügung gestellt.

Das Hauptfenster teilt sich in drei Bereiche. Der größte ist der *Editor-Bereich* links, rechts gibt es oben den *Einstellungs-Bereich* und unten den *Player-Bereich*. Im Editor-Bereich kann über Registerkarten zwischen der Editierung des Keyframe-Netzes (im Folgenden *Bewegungsnetz* bzw. englisch *Motionnet* genannt) und der Editierung des neuronalen Codes gewechselt werden. Wird im Editor-Bereich ein Keyframe oder eine Transition ausgewählt, so werden im Einstellungs-Bereich Regler zur Einstellung der Positionen (bei Keyframes) bzw. der Drehmomente (bei Transitionen) zur Verfügung gestellt. Im Player-Bereich kann ein Motionnet bei angeschlossenem Roboter direkt abgespielt und die Abspielgeschwindigkeit geregelt werden.

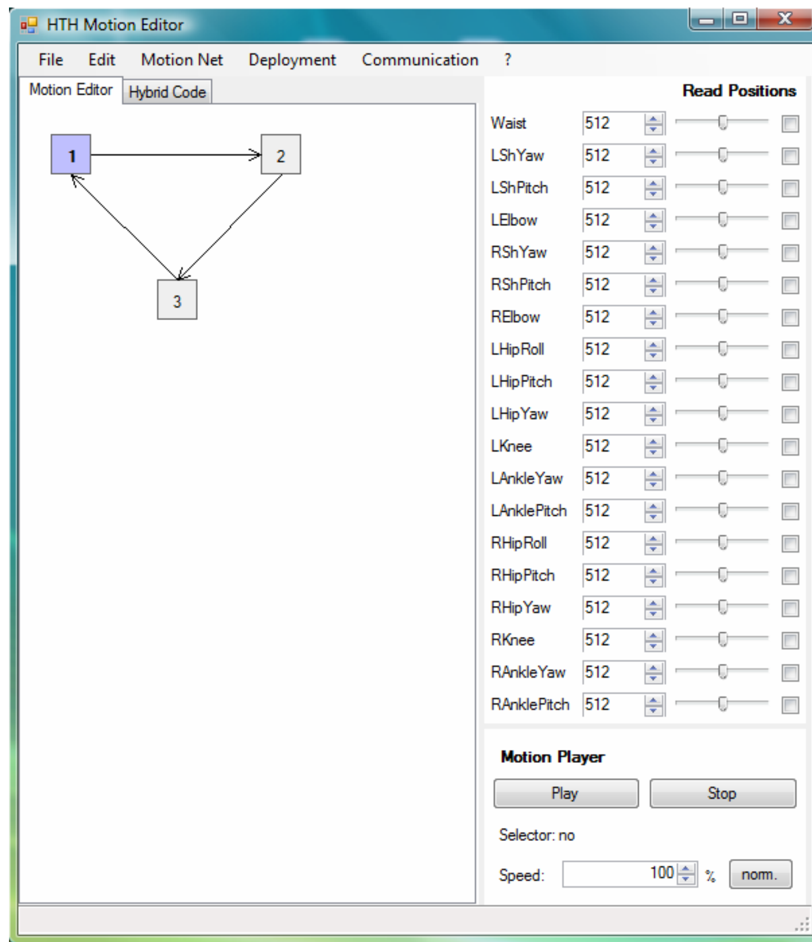


Abbildung 5: Das Hauptfenster des Motion Editors

3.2.1 Erstellung und Manipulation von Motionnets

Der Motion Editor arbeitet bei der Erstellung und Manipulation von Bewegungsnetzen rein grafisch. Keyframes werden als quadratische Blöcke mit einer Keyframe-Nummer dargestellt, Transitionen als Pfeile zwischen Keyframes.

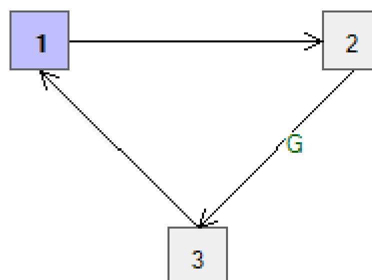


Abbildung 6: Einfaches Motionnet mit drei Keyframes

Abbildung 6 zeigt ein einfaches Motionnet mit drei Keyframes und drei Transitionen. Die Keyframes haben die Nummern 1, 2 und 3; Keyframe 1 ist markiert. Der Transition von Keyframe 2 zu Keyframe 3 ist der Selektor G zugewiesen.

Klickt man mit der rechten Maustaste in eine leere Stelle der Zeichnungsfläche, öffnet sich ein Kontextmenü, über das man ein neues Keyframe anlegen kann, alternativ genügt ein normaler Klick bei gedrückter Strg-Taste. Ein Rechtsklick auf ein Keyframe öffnet wiederum ein Kontextmenü, in dem es die Möglichkeit gibt, eine neue Transition ausgehend von diesem Keyframe anzulegen. Das Transitionsziel wird anschließend per Klick auf das gewünschte Keyframe gewählt. Auch hier gibt es die Alternative, die Transitionsanlage per Klick auf das Startkeyframe bei gleichzeitig gedrückter Strg-Taste zu starten.

Über das Keyframe-Kontextmenü können das Keyframe gelöscht oder die zugehörigen Positionsdaten in die Zwischenablage kopiert werden, um sie in ein anderes Keyframe einzufügen (ebenfalls per dortigem Kontextmenü) oder aus diesen Daten ein neues Keyframe zu erstellen (Kontextmenü der leeren Zeichnungsfläche). Eine weitere Option ist das Spiegeln: Hierbei werden die Positionsdaten entsprechend der Morphologie des Roboters gespiegelt. Auch die Nummer des Keyframes kann geändert werden, wobei jede Nummer nur einmal vergeben werden kann. Die Nummern der Keyframes haben keine Bedeutung und dienen nur der Übersichtlichkeit und der teaminternen Kommunikation über das Bewegungsnetz. Eine Ausnahme bildet das Keyframe mit der Nummer 1, das den Anfang des Motionnets festlegt.

Auch Transitionen verfügen über ein Kontextmenü. Dieses stellt neben dem Löschen der Transition und dem Setzen eines Selektors auch die Möglichkeit zur Verfügung, eine Transition zu teilen. In diesem Fall wird zu einer prozentual gewählten Zwischenzeit ein neues Keyframe so zwischen die beiden vorhandenen Keyframes eingefügt, dass sich am Bewegungsverhalten nichts ändert. Dieses Keyframe kann dann zur Feinanpassung weiter bearbeitet werden. Eine weitere Möglichkeit, die das Transitionen-Kontextmenü bietet ist die Auswahl des Teilprogramms des neuronalen Codes, das bei Abarbeitung dieser Transition ausgeführt werden soll.

Keyframes und Transitionen werden per einfachem Klick selektiert, Keyframes können per Drag & Drop verschoben werden.

Abbildung 7: oberer Ausschnitt der Einstellungen für Keyframes

Wird ein Keyframe ausgewählt, werden im Einstellungs-Bereich entsprechende Optionen angezeigt, wie in Abbildung 7 dargestellt. Zunächst sind das für alle 19 Motoren der Roboter die Positionen, die per Zahleneingabe oder per Schieberegler verändert werden können. Zusätzlich gibt es für jeden Motor die Option „Read Positions". Wird beim Abspielen eines Motionnets ein Keyframe erreicht, bei dem diese Option für den entsprechenden Motor gesetzt ist, wird zum weiteren Interpolieren nicht die angegebene Position sondern die wirklich erreichte Position verwendet.

Ist ein Roboter angeschlossen, wird bei jeder Keyframe-Auswahl und jeder Werteänderung automatisch diese Pose an den Roboter übertragen. So kann man über das Bewegen eines Schiebereglers direkt den entsprechenden Motor steuern. Soll ein Motor von Hand bewegt werden, muss man auf den Namen des entsprechenden Gelenks klicken. Dieser wird dann fett dargestellt und das Gelenk wird freigegeben. Bei erneutem Klick wird die „geknetete" Position ausgelesen und das Gelenk wieder festgestellt. Ein Klick mit aktivierter Shift-Taste gibt ganze Gelenkgruppen (z.B. ein Bein) frei.

Ist ein Keyframe selektiert, können bei gedrückter Shift-Taste per Klick weitere Keyframes hinzugefügt werden. Motoren mit gleichen Positionsdaten werden im Einstellungsbereich normal dargestellt, für solche mit unterschiedlichen Positionsdaten wird der Eintrag grau unterlegt und die Durchschnittsposition angezeigt. Wird ein solcher Wert verändert, wird er für sämtliche selektierte Keyframes übernommen.

Auch für Transitionen werden bei Auswahl entsprechende Einstellungen angezeigt.

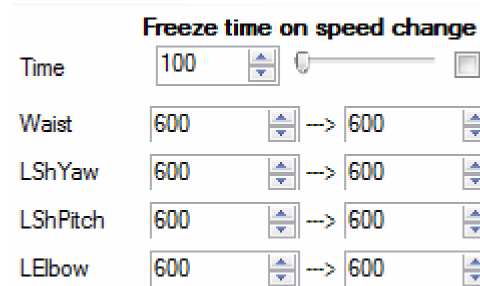


Abbildung 8: oberer Ausschnitt der Einstellungen für Transitionen

Bei Transitionen gibt es zunächst die Möglichkeit, die Übergangszeit einzustellen. Die Einstellung wird in Millisekunden vorgenommen, es sind jedoch nur Vielfache von 10 Millisekunden zugelassen (die Ansteuerung der Motoren auf dem Roboter geschieht mit 100 Hz). Da es im Motion Editor die Möglichkeit gibt, die Abspielgeschwindigkeit anzupassen, kann man hier auch festlegen, dass dies für die gewählte Transition nicht geschehen soll. Diese Einstellung hat keine Auswirkung auf den autonomen Betrieb des Roboters.

Unterhalb der Zeiteinstellung können für jeden Motor die Drehmomentangaben konfiguriert werden. Da – wie bereits erwähnt – bei diesen Daten Sprünge beim Durchlaufen eines Keyframes möglich sind, wird jeweils pro Transition ein Start- und ein Enddrehmoment angegeben. Dazwischen wird – wie bei den Positionsdaten auch – linear interpoliert. Am unteren Ende gibt es die Zeile „All Motors", die es ermöglicht, die Drehmomente für alle Motoren gleichzeitig zu setzen.

Auch bei Transitionen ist es möglich, mehrere auszuwählen. Es gilt dabei das gleiche, was bereits für Keyframes beschrieben wurde.

Abbildung 9 auf der nächsten Seite zeigt einen Ausschnitt aus einem komplexen Motionnet. Dargestellt wird auch die Möglichkeit, Bewegungsgruppen (beispielsweise das „Laufen" links oben) mit sogenannten „Labels" farblich zu markieren und zu gruppieren. Eine weitere Möglichkeit ist das Importieren von Bewegungsnetzen (z.B. Teilbewegungen) in andere Bewegungsnetze.

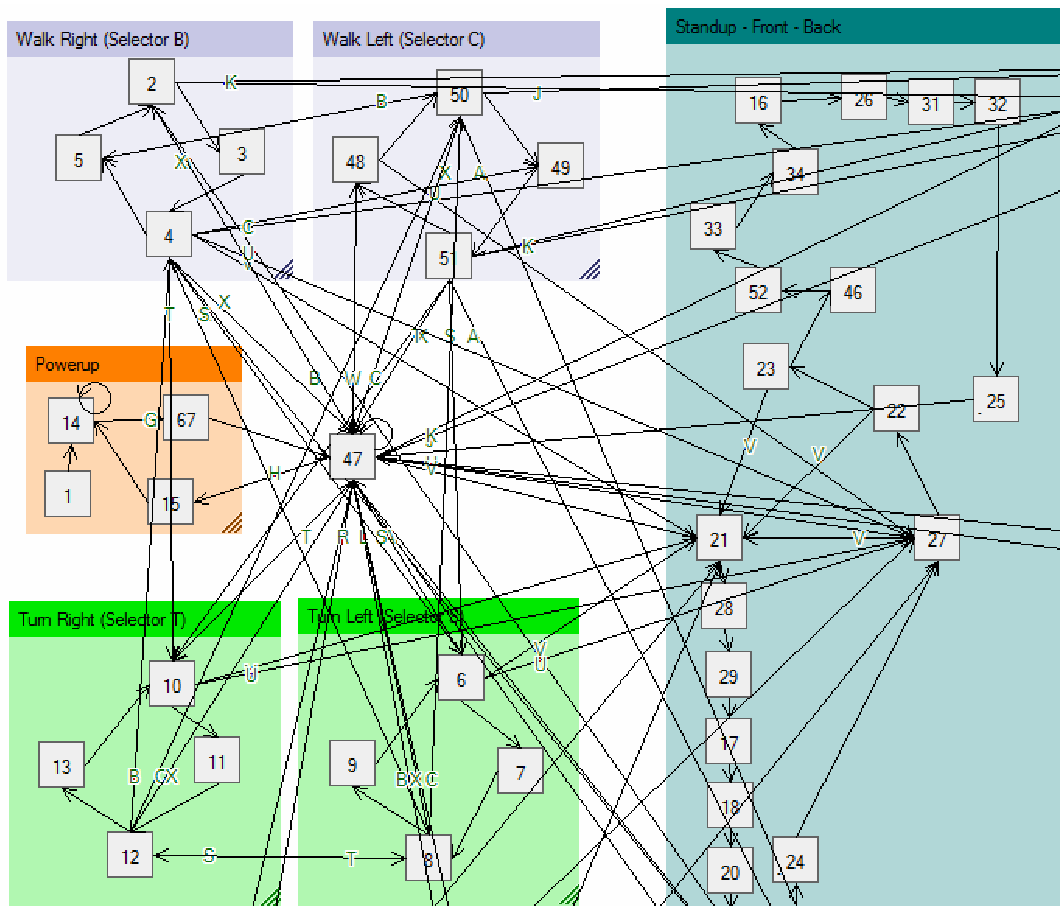


Abbildung 9: Komplexes Motionnet

3.2.2 Bewegungen abspielen

Im unteren rechten Bereich des Motion Editors befindet sich der *Player*. Hier ist es möglich, ein gerade geöffnetes Bewegungsnetz bei angeschlossenem Roboter abzuspielen. Zu Justagezwecken kann die Geschwindigkeit des Abspielens prozentual angepasst werden. Sobald sich eine andere Grundgeschwindigkeit (beispielsweise halb so schnell) als sinnvoll erweist, können alle Transitionszeiten entsprechend auf Knopfdruck angepasst werden. Wie genau die Verbindung mit dem Roboter funktioniert wird im Kapitel *4.4 Kommunikation zwischen Entwicklungsumgebung und Roboter* auf Seite 22 erläutert.

3.2.3 Editor für neuronalen Code

Über Registerkarten oberhalb des Editor-Bereichs kann der Editor für den neuronalen Code aufgerufen werden. Es handelt sich um einen Texteditor mit Syntax Highlighting, also farblicher Darstellung von erkannten Befehlen und Kommentaren. Der Übersetzer für den Code ist direkt in den Motion Editor integriert und meldet bei Fehlern die entsprechenden Zeilennummern, die dann grafisch hervorgehoben werden. Dies soll die Erstellung des Codes erleichtern.

3.2.4 Weitere Funktionen

Der Motion Editor kann natürlich Bewegungsnetze vollständig speichern. Zusätzlich zum Motionnet können in der sogenannten „.mef“-Datei („mef“ für Motion Editor File) Informationen zum Netz abgespeichert werden, beispielsweise wer das Netz erstellt hat und wann es für welchen Roboter angepasst wurde. Auch weitergehende Kommentare können direkt in der Datei abgelegt werden. Der neuronale Code wird in einer zusätzlichen, verknüpften Datei abgespeichert.

Über den Austausch dieser „mef“-Dateien ist auch der Austausch mit *Simloid* [Hei07], der Simulationsumgebung des Humanoid Team Humboldt, gewährleistet. Simloid kann „mef“-Dateien importieren und so die Bewegungen im Simulator abspielen. Auf diese Weise ist es möglich, auch schnelle Bewegungen genau und in Zeitlupe zu analysieren. Umgekehrt kann Simloid in Simulationen gewonnene Bewegungen als „mef“-Dateien exportieren. Dabei kann Simloid derzeit noch keine neuronale Steuerung exportieren, die gewonnenen Bewegungen sehen nach Übertragung auf echte Hardware (über den Motion Editor) zwar ähnlich aus, funktionieren jedoch aufgrund der Unterschiede zwischen realer Welt und Simulation derzeit noch nicht.

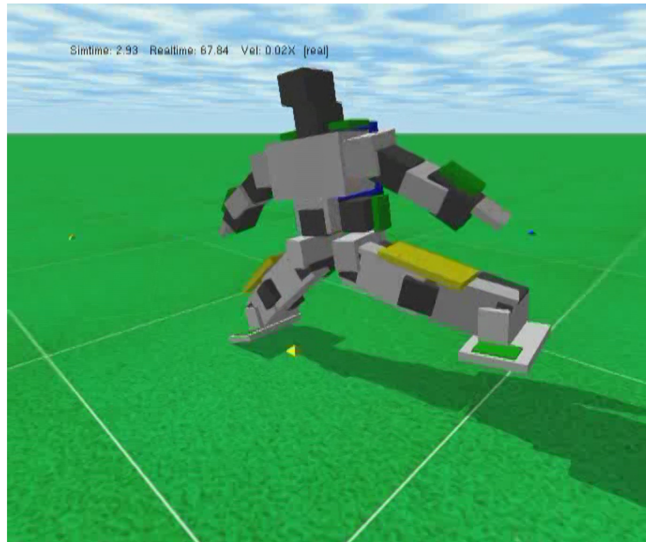


Abbildung 10: Der Simulator Simloid spielt eine Bewegung ab, bei welcher der Torwart die Beine zur Abwehr des Balls spreizt. Diese Bewegung wurde im Motion Editor entwickelt und konnte später im Simulator in Zeitlupe analysiert werden.

Eine während der Bewegungserstellung unerlässliche Funktion ist das sogenannte „Undo“: letzte, fehlerhafte Änderungen können rückgängig gemacht werden. Das Überspielen der Bewegungsdaten auf den Roboter wird im Kapitel 4.3 auf Seite 21 erläutert.

3.3 Evolutionäre Entwicklung

Die Entwicklung des Motion Editors startete Ende März 2006 unter der Maßgabe, möglichst innerhalb weniger Wochen ein lauffähiges, einsatzfähiges Programm zur Verfügung zu haben, da die Bewegungserstellung für den Betrieb der Roboter natürlich unentbehrlich ist. Es wurde also eine evolutionäre Entwicklung vorgesehen, die mit einem schnell entwickelten Prototyp beginnen sollte, der dann nach und nach den Erfordernissen angepasst werden sollte. Als Implementierungssprache wurde C# gewählt, da diese es erlaubt, schnell grafische Oberflächen zu entwickeln. Ein weiterer Punkt war die durch die Mono-Implementierung des .NET-Frameworks gegebene Betriebssystemunabhängigkeit, die jedoch später aufgrund der Bindung an eine Treiber-DLL zur Kommunikation mit den Robotern aufgegeben werden musste. Der Motion Editor ist also nur unter dem Betriebssystem Windows lauffähig, eine denkbare, abgespeckte Version ohne direkte Roboterkommunikation wäre jedoch auch auf anderen Systemen (inklusive Linux, Solaris und MacOS X) lauffähig.

So konnte ich die Grundfunktionalität mit dem Anlegen von Keyframes und Transitionen, dem Speichern und Öffnen von Bewegungsnetzen und dem Abspielen aus dem Motion Editor heraus innerhalb nur einer Woche realisieren. Innerhalb einer weiteren Woche wurden erste im Einsatz gefundene Probleme behoben und das Übertragen der Netze auf den Roboter ermöglicht. Nach nur zwei Wochen war es also möglich, Bewegungsnetze zu erstellen und diese auf dem Roboter anzuwenden.

Änderungen resultierten dann in der Folgezeit zunächst vorrangig aus den Hinweisen der Benutzer. So wurden die Labels und die „Undo“-Funktion auf Bitte von Anwendern implementiert. Bis Anfang Juni 2006 war das Programm größtenteils fertig.

Was nun zum weiteren Ausbau fehlte waren Funktionen der AccelBoards. So war es nicht möglich, die Bewegungsnetze per Mausklick auf die AccelBoards zu verteilen: Der Motion Editor erstellte stattdessen eine Datei, die in das Programm eines der AccelBoards integriert und manuell auf dieses aufgespielt werden musste. Neuronaler Code war auf diese Weise nicht auf alle AccelBoards zu verteilen und von diesen zu berechnen, da die Bewegungssteuerung von nur einem der AccelBoards übernommen wurde. Es folgte eine längere Implementierungsphase aufseiten der AccelBoards.

Hier wurde ein verteiltes System implementiert, bei dem im 100-Hz-Rhythmus alle AccelBoards Daten austauschen und die Motoren ansteuern. Über denselben Bus wurde später das Verteilen der Bewegungsdaten („Deployment“, siehe Kapitel 4.3) realisiert. Nach der Umstellung der AccelBoards auf dieses neue System waren auch Anpassungen aufseiten des Motion Editors nötig, so musste die Ansteuerung des Roboters nun anders gelöst werden, um nicht mit dem Austauschprotokoll der AccelBoards zu kollidieren (mehr dazu in Kapitel 4.4).

4 Hardware-Plattform

4.1 Die Roboter des Humanoid Team Humboldt

Die humanoiden Roboter des Humanoid Team Humboldt der Serie A (HTH/A) basieren auf dem Bioloid-Bausatz der koreanischen Firma Robotis. Die 42 cm großen und 2,1 kg schweren Roboter besitzen insgesamt 21 Freiheitsgrade, davon 19 zur Bewegungssteuerung des Körpers und zwei weitere zur Steuerung des Kopfes. Dieser wird jedoch nicht vom hier beschriebenen System gesteuert.

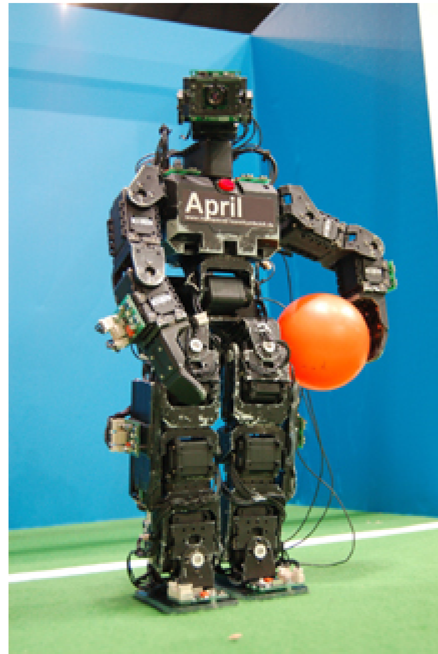


Abbildung 11: Roboter „April“

Erweitert wurde das System um die acht bereits erwähnten AccelBoards, auf die im nächsten Kapitel detaillierter eingegangen wird. Der mitgelieferte Steuerungsprozessor samt Stromversorgung wurde entfernt und als Ersatz wurden drei leistungsstarke Lithium-Polymer-Akkus am Körper angebracht. Auf dem Rücken wurde ein handelsüblicher PDA der Firma Fujitsu-Siemens („Pocket Loox“) montiert, der von einer auf dem Kopf angebrachten Kamera Bildsignale erhält, diese analysiert und über ein Verhalten die gewünschten Aktionen an die AccelBoards meldet. Die Roboter werden derzeit vorrangig zum Roboterfußball im Rahmen des RoboCup [BM03] eingesetzt.

4.2 Steuerungsplatinen der Roboter

An den Robotern sind jeweils acht sogenannte AccelBoards platziert. Diese kommunizieren über einen gemeinsamen Bus, den sogenannten *SpinalCord*[3], miteinander. Dieser ist ein auf RS485 basierender asynchroner Half-Duplex-Bus mit einer Geschwindigkeit von einem MBaud.

Jedes der Boards ist rund 3,8 mal 2,6 cm groß und besitzt einen 16-Bit-RISC-Prozessor (Renesas R8C/11) und einen 2-Achsen-Beschleunigungssensor mit einer 16-Bit-Auflösung für beide Achsen (Analog Devices ADXL213). Zusätzlich befinden sich auf dem Board neben einem Spannungsregler Anschlüsse für die Busse (über die auch der Prozessor programmiert werden kann), zwei Taster sowie zwei LEDs, die vom Prozessor zur Anzeige des Zustands verwendet werden.

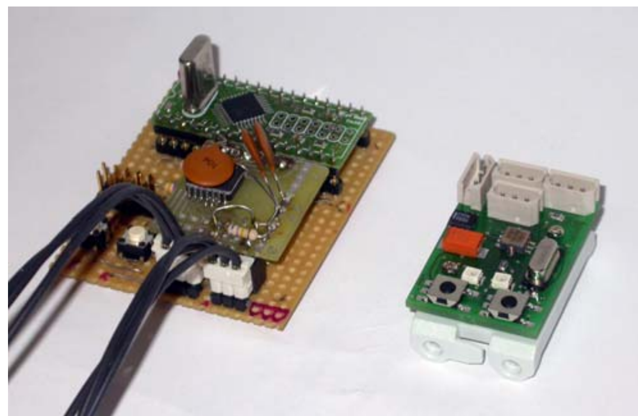


Abbildung 12: Prototyp eines AccelBoards (links) und ein fertiges AccelBoard montiert auf einem Bioloid-Versatzstück (rechts)

Eines der Boards übernimmt die *Master*-Rolle und gibt den Zeitrahmen für die Kommunikation über den SpinalCord vor. Mit diesem Board kommuniziert auch die Verhaltenssoftware auf dem PDA und teilt beispielsweise mit, welche Bewegung gewünscht ist.

[3] Engl. für „Rückenmark“

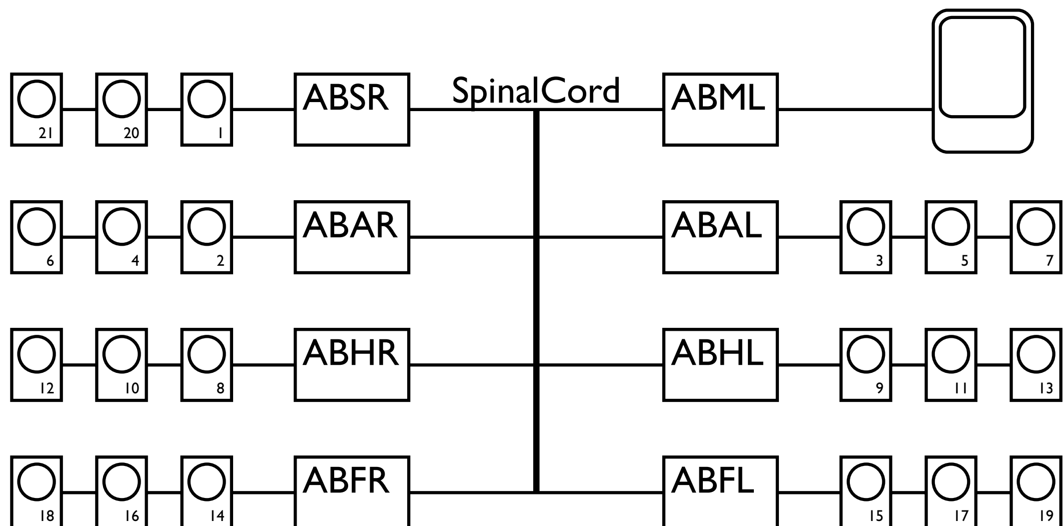


Abbildung 13: Verteilung und Aufgaben der AccelBoards und der Motoren

Die anderen sieben Boards sind sogenannte *Slaves*. An jedem der Slaves hängen drei Motoren, die von diesem gesteuert werden. Sensorinformationen (neben den Beschleunigungsdaten auch ausgelesene Motorpositionen) werden über den SpinalCord mit den anderen Boards kommuniziert.

4.3 Autonomer Betrieb der Roboter

Die Grundstruktur der Software für die Boards wurde zunächst vom Teamchef des Humanoid Team Humboldt, Manfred Hild, entwickelt und anschließend von mir um die für die Bewegungssteuerung nötigen Teile ergänzt. Die Software wurde vollständig in C und Assembler entwickelt, wobei aus Performancegründen ein relativ hoher Anteil von über 35% in Assembler implementiert wurde. Um AccelBoards schnell und unkompliziert austauschen zu können, läuft auf sämtlichen Boards die identische Software – an welcher Stelle das Board montiert ist, erkennt dieses über die angeschlossenen Motoren (die jeweils eindeutige IDs besitzen).

Zunächst jedoch implementierte Manfred Hild im April 2006 parallel zu meiner Entwicklung des Motion Editors einen Player für die in Kapitel 2.2 beschriebene Keyframe-Methode, der nicht auf der verteilten Architektur basierte. Hier steuerte das Master-Board sämtliche Motoren, die anderen Boards leiteten die Daten einfach nur weiter. Über ein vorher abgestimmtes Datenformat wurden die Bewegungsdaten aus dem Motion Editor exportiert und direkt in die Software einkompiliert. Die Benutzung

von drei verschiedenen Windows-Softwares (Motion Editor; Compiler; Flasher zum Überspielen auf die Boards) machte diesen Arbeitsschritt zeitintensiv.

Erst die automatische Verteilung der Bewegungsdaten auf alle angeschlossenen Accel-Boards („Deployment") erlaubt die getrennte Steuerung von jeweils drei Motoren durch ein Board und damit verbunden eine neuronale Nachsteuerung und den Austausch von Sensorinformationen über den SpinalCord. Für dieses Deployment wird der PC über USB an eine vom Team selbst entwickelte Box (genannt „MatchBox") angeschlossen, die am anderen Ende direkt an den SpinalCord gehängt wird. Über einen Befehl vom PC löschen alle Boards ihren Datenspeicher und nehmen dann Datenpakete entgegen, die diese direkt in diesen Speicher schreiben. Auf PC-Seite wird dies vom Kommandozeilen-Programm *MotionDeployer* (implementiert in C++) realisiert, das jedoch direkt aus dem Motion Editor heraus aufgerufen wird und dessen Ausgaben ebenfalls im Motion Editor angezeigt werden.

Die Steuerung auf den Boards übernimmt ein nur wenig abgewandelter Keyframe-Player. Anschließend können diese Daten durch neuronalen Code (wie in Kapitel 2.4 beschrieben) manipuliert werden. Der in Anhang 1 beschriebene neuronale Assembler wird dabei in einen Bytecode überführt, bei dem jeder Befehl mitsamt Parametern genau 16 Bit einnimmt. Der entsprechende Übersetzer auf PC-Seite ist direkt in den Motion Editor integriert, liegt jedoch auch als betriebssystemunabhängige Kommandozeilenversion vor. Der so erstellte Bytecode wird genau wie die Keyframes und Transitionen direkt in den Speicher der Boards „deployt". Die Abarbeitung dort wurde aus Performancegründen vollständig in Assembler implementiert.

4.4 Kommunikation zwischen Entwicklungsumgebung und Roboter

Ein wichtiger Punkt bei der Entwicklung des Motion Editors war, dass dieser direkt am Roboter betrieben werden sollte. Zunächst wurde auch hier eine Übergangslösung implementiert, um möglichst schnell arbeiten zu können. Dabei wurde das Master-Board abgesteckt und durch ein spezielles Board ersetzt, das die Bewegungsanforderungen des Motion Editors ausführte.

Wichtig ist dabei, dass sämtliche Posen sofort am Roboter sichtbar sind und auch direkt an diesem „geknetet" werden können. Da es durch die Verteilung der Steuerung auf acht Boards nicht mehr möglich war, direkt alle Motoren extern zu steuern, wurde der sogenannte „Transparent Mode" eingeführt. Zu definierten, kommunikationsfrei-

en Zeiten auf dem SpinalCord kann der PC so gewünschte Posen übermitteln, die die einzelnen Boards dann ausführen. Die Kommunikation zwischen den Boards findet trotzdem noch statt und der PC kann diese auswerten und so Motorpositionen nach dem „Kneten“ auslesen.

Da die Kommunikation auf dem SpinalCord zeitgenau geschehen muss und dafür eine Treiber-DLL des Herstellers FTDI nötig war, wurde diese Ansteuerung in eine Extra-DLL, die in C++ entwickelt wurde, ausgelagert.

5 Zusammenfassung und Ausblick

Im Rahmen dieser Studienarbeit wurde die Windows-Software zur Erstellung von Bewegungen für die humanoiden Roboter des Humanoid Team Humboldt entwickelt. Die Steuerung, basierend auf der Keyframe-Methode, ist bereits seit anderthalb Jahren im produktiven Einsatz. Bewegungsnetze wurden von verschiedenen Team-Mitgliedern erstellt und wurden später autonom auf den Robotern – beispielsweise bei RoboCup-Wettkämpfen – ausgeführt. Die schrittweise Entwicklung der Software erlaubte den schnellen ersten Einsatz und anschließend die zeitintensive Entwicklung des verteilten Steuerungssystems mit Abwärtskompatibilität. Nun ist es mithilfe des entwickelten neuronalen Bytecodes möglich, einzelne Bewegungen neuronal zu stabilisieren oder Bewegungen (wie das Laufen) vollständig neuronal auf den Robotern auszuführen.

Dabei konnte die Software neben einigen Komfortfunktionen mit allen wichtigen Funktionen zur Bewegungssteuerung ausgestattet werden: Keyframes können geknetet werden, Bewegungen direkt aus dem Motion Editor heraus abgespielt werden und anschließend per Knopfdruck auf den Roboter übertragen werden, sodass dieser sich autonom bewegen kann.

Bisher wurde der zuletzt implementierte neuronale Bytecode nur testweise genutzt, es steht nun die Nutzung für echte neuronale Netze zur Stabilisierung oder zum Laufen an, wie sie von Team-Mitgliedern schon vom PC aus entwickelt wurden. Die Ausführung der bereits von Hand erstellten Netze direkt auf dem Roboter sollte dessen Bewegungen verbessern. Auf der anderen Seite wird derzeit ein verbesserter Simulator entwickelt, in dem evtl. in absehbarer Zeit neuronale Netze per künstlicher Evolution erstellt werden können, die – per neuronalem Bytecode auf den Roboter übertragen – noch bessere Ergebnisse erzielen. Denkbar wäre an dieser Stelle, im Simulator den gleichen neuronalen Bytecode zu benutzen und evtl. per genetischer Programmierung (siehe Kapitel 2.1) interessante Ergebnisse zu erzielen.

Anhang 1: Neuronaler Bytecode

Zur neuronalen Manipulation von Bewegungsdaten wurde in Kooperation mit der Universität Osnabrück der im Folgenden beschriebene neuronale Bytecode mit zugehöriger Assemblersprache entwickelt und implementiert. Er ist bewusst allgemein gehalten und wird nicht nur auf unseren humanoiden Robotern der Serie HTH/A eingesetzt, sondern wird derzeit auch für die achtbeinige Laufmaschine *Octavio* der Universität Osnabrück implementiert.

Es wird zunächst vorausgesetzt, dass sämtliche Sensordaten und in unserem Fall auch zukünftig anzusteuernden Werte (aus dem Keyframe-Player) in einem Feld mit 16-Bit-Werten abgelegt sind (die 16 Bit sind eine Einschränkung des Interpreters auf unseren AccelBoards, nicht der Sprache an sich). Dieses entspricht genau den Daten, die über den gemeinsamen Bus der AccelBoards (den *SpinalCord*) ausgetauscht werden, sie werden daher auch als SpinalCord-Array oder SpinalCord-Daten bezeichnet. Diese 16-Bit-Werte werden – wie alle neuronalen Eingänge und Ergebnisse dieses Interpreters – als Festkomma-Werte zwischen +1 (ausschließlich) und –1 (einschließlich) interpretiert. In unserem Fall besitzt jedes Board einen Slot mit 16 Werten, insgesamt also 128 Werte, die von 0 bis 127 adressiert werden können. In den 16 Werten bringt jedes Slave-Board neben den Ansteuerungs- und Sensordaten einige freie Felder unter, die zum Austausch benutzt werden können. Zusätzlich stehen interne Neuronenspeicher zur Verfügung, auf unseren Boards derzeit jeweils 32. Die Sprache kann bis zu 2048 SpinalCord-Werte und bis zu ebenfalls 2048 Neuronen-Werte adressieren.

Wie bei neuronalen Netzen üblich, sollte die Reihenfolge der Neuronen keine Rolle spielen, die entsprechenden Neuronenausgänge dürfen erst nach einem Zeitschritt mit den neuen Werten aktualisiert werden. Daher existieren neben den Neuronen und den SpinalCord-Daten jeweils noch die gepufferten Äquivalente, die nach der Berechnung eines Zeitschrittes zurückgespielt werden. Implementierungsgebunden wird in unserem Fall immer nur der das Board betreffende Slot des SpinalCords (dessen 16 Werte) gepuffert.

Der neuronale Assembler arbeitet auf drei Registern. Das W-Register speichert ein Gewicht (13-Bit-Festkomma mit vier Vorkomma-Bit), das V-Register einen Wert aus einem Neuron oder dem SpinalCord-Array (16-Bit-Festkomma mit einem Vorkom-

ma-Bit/Vorzeichen-Bit). Der ACCU ist 32 Bit breit und hält Zwischenergebnisse (8 Bit Vorkomma, 24 Bit Nachkomma).

Die Sprache besitzt acht Haupt-Befehle (adressiert über die ersten drei Bit), sechs dieser Befehle werden über zwei weitere Bit zu jeweils vier Unterbefehlen erweitert. Die Befehle werden auf den folgenden Seiten einzeln aufgeführt. Dabei wird zusätzlich für jeden Befehl die in unserer Implementierung zur Abarbeitung benötigten Zyklen des verwendeten R8C/11-Prozessors angegeben. Der Programmstart benötigt einmalig 191 Zyklen. Die Zyklen geben jeweils die längstmögliche Laufzeit an, hinzu kommen pro Befehl 21 Zyklen zur Auswahl des entsprechenden Codes.

weight

Der weight-Befehl speichert das übergebene Gewicht (13 Bit, 4 Vorkomma-Bit, von einschließlich –8 bis ausschließlich +8) im Register W.

Bytecode (V = Vorkomma; N = Nachkomma):

0	0	0	V	V	V	V	N	N	N	N	N	N	N	N	N

Kurzform:

```
W <- value
```

Assembler:

```
weight <value>
```

Prozessor-Zyklen: 12

read

Der read-Befehl liest aus dem SpinalCord oder den Neuronen einen Wert in das V-Register. Er unterteilt sich in vier Unterbefehle. Diese geben an, von wo gelesen werden soll. Die vier Möglichkeiten sind: SpinalCord, Neuron, SpinalCord-Buffer und Neuron-Buffer. Im Assembler-Code gibt es dabei Alias-Bezeichnungen für die ungepufferten Versionen, da diese im Normalfall verwendet werden. Aus den Puffern wird normalerweise nicht gelesen. Die folgenden 11 Bit geben als Ganzzahl ohne Vorzeichen die jeweilige Position an.

read.s

Bytecode:

0	0	1	0	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
V <- SpinalCord[A]
```

Assembler:

```
read.s <A>
read <A>
```

Prozessor-Zyklen: 13

read.n

Bytecode:

0	0	1	0	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
V <- Neuron[A]
```

Assembler:

```
read.n <A>
read N<A>
```

Prozessor-Zyklen: 13

read.sb

Bytecode:

0	0	1	1	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
V <- SpinalCordBuffer[A]
```

Assembler:

```
read.sb <A>
```

Prozessor-Zyklen: 13

read.nb

Bytecode:

0	0	1	1	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
V <- NeuronBuffer[A]
```

Assembler:

```
read.nb <A>
```

Prozessor-Zyklen: 13

write

Der write-Befehl schreibt aus dem V-Register in den SpinalCord oder die Neuronen. Er unterteilt sich in vier Unterbefehle. Diese geben an, wohin geschrieben werden soll. Die vier Möglichkeiten sind: SpinalCord, Neuron, SpinalCord-Buffer und Neuron-Buffer. Die folgenden 11 Bit geben als Ganzzahl ohne Vorzeichen die jeweilige Position an. Dieser Befehl wird nur in Verbindung mit dem read-Befehl zum Kopieren von Daten benutzt. In unserer Implementierung geht dies mit einem Befehl, dem copy-Befehl.

write.s

Bytecode:

0	1	0	0	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
SpinalCord[A] <- V
```

Assembler:

```
write.s <A>
```

Prozessor-Zyklen: 13

write.n

Bytecode:

0	1	0	0	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
Neuron[A] <- V
```

Assembler:

```
write.n <A>
```

Prozessor-Zyklen: 13

write.sb

Bytecode:

0	1	0	1	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
SpinalCordBuffer[A] <- V
```

Assembler:

```
write.sb <A>
write <A>
```

Prozessor-Zyklen: 13

write.nb

Bytecode:

0	1	0	1	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
NeuronBuffer[A] <- V
```

Assembler:

```
write.nb <A>
write N<A>
```

Prozessor-Zyklen: 13

calc

Der calc-Befehl wird zum Ausführen einer Rechnung verwendet. Er unterteilt sich in vier Unterbefehle, von denen zwei jedoch derzeit nicht verwendet werden. Auch die übrigen elf Bit sind derzeit unbenutzt. `calc.m` bzw. `mac` multiplizieren das Register V mit dem Register W und addieren es auf das ACCU-Register. Auf diese Weise wird ein Neuroneneingang auf die Aktivierung im ACCU-Register addiert. `calc.b` bzw. `bias` addiert nur den Wert des W-Registers auf den ACCU und kann somit als Bias-Term verwendet werden.

calc.m

Bytecode:

0	1	1	0	0	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
ACCU <- ACCU + V * W
```

Assembler:

```
calc.m
mac
```

Prozessor-Zyklen: 20

calc.b

Bytecode:

0	1	1	0	1	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
ACCU <- ACCU + W
```

Assembler:

```
calc.b
bias
```

Prozessor-Zyklen: 20

act

Der act-Befehl ist für Aktivierungsfunktionen der Neuronen gedacht. Er wird auf den ACCU angewendet. Er unterteilt sich in bis zu vier verschiedene Aktivierungsfunktionen, die folgenden elf Bit werden derzeit ignoriert. Die festgelegten Aktivierungsfunktionen sind: act.t (tanh), act.s (Standardsigmoide) und act.l (lineare Funktion). Auf unseren Robotern wird ausschließlich der Tangens Hyperbolicus verwendet, die anderen Funktionen wurden nicht implementiert. Unsere Lookup-Table-Implementierung mit 41 Stützstellen erreicht dabei im Interval [-5;+5] eine Genauigkeit von durchschnittlich 0,055%. Der höchste Fehler nah bei 0 liegt bei bspw. ±0,0001 bei 0,52%.

act.t

Bytecode:

1	0	0	0	0	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
ACCU <- tanh(ACCU)
```

Assembler:

```
act.t
tanh
```

Prozessor-Zyklen: 56

act.s

Bytecode:

1	0	0	0	1	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
ACCU <- sigm(ACCU)
```

Assembler:

```
act.s
sigm
```

Prozessor-Zyklen: nicht implementiert

act.l

Bytecode:

1	0	0	1	0	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
ACCU <- lin(ACCU)
```

Assembler:

```
act.l
lin
```

Prozessor-Zyklen: nicht implementiert

res

Der res-Befehl (von engl. „result“) speichert ein Ergebnis aus dem ACCU an die Zielposition im SpinalCord oder einem Neuron. Er unterteilt sich in vier Unterbefehle. Diese geben an, wohin geschrieben werden soll. Die vier Möglichkeiten sind: SpinalCord, Neuron, SpinalCord-Buffer und Neuron-Buffer. Im Assembler-Code gibt es dabei Alias-Bezeichnungen für die gepufferten Versionen, da diese im Normalfall verwendet werden. In die ungepufferten Felder wird meist nicht geschrieben, da diese nach Beendigung normalerweise von den gepufferten Feldern überschrieben werden. Die folgenden 11 Bit geben als Ganzzahl ohne Vorzeichen die jeweilige Position an. Bei jedem res-Befehl wird dabei automatisch der ACCU geleert.

res.s

Bytecode:

1	0	1	0	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
SpinalCord[A] <- ACCU
ACCU <- 0
```

Assembler:

```
res.s <A>
```

Prozessor-Zyklen: 49

res.n

Bytecode:

1	0	1	0	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
Neuron[A] <- ACCU
ACCU <- 0
```

Assembler:

```
res.n <A>
```

Prozessor-Zyklen: 49

res.sb

Bytecode:

1	0	1	1	0	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
SpinalCordBuffer[A] <- ACCU
ACCU <- 0
```

Assembler:

```
res.sb <A>
res <A>
```

Prozessor-Zyklen: 49

res.nb

Bytecode:

1	0	1	1	1	A	A	A	A	A	A	A	A	A	A	A

Kurzform:

```
NeuronBuffer[A] <- ACCU
ACCU <- 0
```

Assembler:

```
res.nb <A>
res N<A>
```

Prozessor-Zyklen: 49

copy

Der Copy-Befehl dient zum Kopieren aus dem Neuronen-Puffer in den SpinalCord-Puffer. Dazu werden nach den drei Befehls-Bits sechs Bits für die Neuronen-Position und sieben Bits für die SpinalCord-Position angegeben. Der Befehl ist damit auf Implementierungen beschränkt, deren SpinalCord maximal 127 Felder hat und die mit maximal 64 Neuronen arbeiten. Er lässt sich jedoch durch die Hintereinanderausführung eines read.nb und eines write.sb-Befehls simulieren.

Bytecode:

1	1	0	N	N	N	N	N	N	S	S	S	S	S	S	S

Kurzform:

```
SpinalCordBuffer[S] <- NeuronBuffer[N]
```

Assembler:

```
copy <N> <S>
```

Prozessor-Zyklen: 28

eoc

Der letzte Befehlsblock ist für EOC („End of Code“, Ende des Codes) vorgesehen. Sämtlicher nicht benutzter Speicherbereich wird dabei mit dem Bit 1 gefüllt. Festgelegt sind dabei aber nur die ersten fünf Bits, die restlichen können beliebig sein. Der End of Code muss im Assemblerprogramm nicht angegeben werden, er wird implizit ans Ende gestellt und hat hauptsächlich im Bytecode eine Bedeutung. Zusätzlich gibt es die Möglichkeit des eoc.no_copy, bei dem die gepufferten Werte von SpinalCord und Neuronen nicht zurückkopiert werden.

eoc.no_copy

Bytecode:

1	1	1	0	0	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
nop
```

Assembler:

```
eoc.no_copy
eoc.nc
```

Prozessor-Zyklen: 6

eoc

Bytecode:

1	1	1	1	1	?	?	?	?	?	?	?	?	?	?	?

Kurzform:

```
Neuron <- NeuronBuffer
SpinalCord <- SpinalCordBuffer
```

Assembler:

```
eoc
```

Prozessor-Zyklen: 481

Literatur

[Bae69] Ronald M. Baecker: *Picture-driven animation*. Proceedings AFIPS 1969 Spring Jt. Computer Conference, Vol. 34, AFIPS Press, Montvale, New Jersey, Seiten 273–288, 1969

[BM03] Hans-Dieter Burkhard, Hans-Arthur Marsiske: *Endspiel 2050*. Heise, 2003.

[Bra84] Valentino Braitenberg: *Vehicles. Experiments in Synthetic Psychology*. MIT Press, Cambridge, Massachusetts, 1984

[BS79] Norman I. Badler, Stephen W. Smoliar: *Digital Representations of Human Movement*. Computing Surveys, Volume 11, Issue 1, Seiten 19–38, 1979

[BZA02] Jens Busch, Jens Ziegler, Christian Aue, Andree Ross, Daniel Sawitzki, Wolfgang Banzhaf: *Automatic Generation of Control Programs for Walking Robots Using Genetic Programming*. Proceedings of the 4th European Conference on Genetic Programming, EuroGP 2002, Ausgabe 2278 der Lecture Notes in Computer Science, Seiten 258–268, Springer, New York, 2002

[Hei07] Daniel Hein: *Simloid: Evolution of Biped Walking Using Physical Simulation*. Diplomarbeit, Institut für Informatik, Humboldt-Universität zu Berlin, 2007

[Hil07] Manfred Hild: *Neurodynamische Module zur Bewegungssteuerung autonomer mobiler Roboter*. Dissertation, Institut für Informatik, Humboldt-Universität zu Berlin, 2007

[KB03] Yoshihiro Kuroki, Bill Blank, Tatsuo Mikami, Patrick Mayeux, Atsushi Miyamoto, Robert Playter, Kenichiro Nagasaka, Marc Raibert, Masakuni Nagano, Jin'ichi Yamaguchi: *Motion Creating System For a Small Biped Entertainment Robot*. Proceedings of the 2003 IEEE/RSJ Intl. Conference on Intelligent Robots and Systems, Seiten 1394–1399, 2003

[Mem03] Aydemir Memisoglu: *Human Motion Control using Inverse Kinematics*. Master-Thesis, Institute of Engineering and Science, Bilkent University, 2003

[NF00] Stefano Nolfi, Dario Floreano: *Evolutionary Robotics*. MIT Press, Cambridge, Massachusetts, 2000

[SB85] Scott N. Steketee, Norman I. Badler: *Parametric Keyframe Interpolation incorporating Kinetic Adjustment and Phrasing Control*. International Conference on Computer Graphics and Interactive Techniques / Proceedings of the 12th annual conference on Computer graphics and interactive techniques, Seiten 255–262, 1985

[Tha89] D. Thalmann: *Motion Control: from Keyframe to Task-Level Animation*. State-of-the-Art in Computer Animation, Springer, Seiten 3-17, 1989

[WH04] Takenori Wama, Masayuki Higuchi, Hajime Sakamoto, Ryohei Nakatsu: *Realization of Tai-chi Motion Using a Humanoid Robot*. Proceedings of The 14th International Conference on Artificial Reality and Telexistence, Seiten 71–74, 2004

[YI06] Toshihiko Yanase, Hitoshi Iba: *Evolutionary Motion Design for Humanoid Robots.* Proceedings of the 8th annual conference on Genetic and evolutionary computation (Genetic And Evolutionary Computation Conference), Seiten 1825–1832, 2006