Universität Hamburg

DER FORSCHUNG | DER LEHRE | DER BILDUNG

Fakultät für Mathematik, Informatik und Naturwissenschaften (MIN)
Fachbereich Informatik

Bachelorarbeit

Entwicklung einer Benutzerschnittstelle zur Echtzeitbearbeitung von Roboteranimationen

Development of a user-interface for realtime editing of robot animations

Autor: Timon Giese
timon.giese@informatik.uni-hamburg.de

Matrikelnummer 5944949

Erstkorrektor Dr. Andreas Mäder

Zweitkorrektor Bernd Schütz

Datum 31. August 2015

Kurzfassung

Diese Arbeit behandelt den Entwurf und die Realisierung einer anwenderfreundlichen Anwendung zum Erstellen und Bearbeiten von statischen Roboteranimationen (auch engl. *Motion* bzw. Bewegungen) in Echtzeit direkt auf einem Roboter zur Ausführung in einem beliebigen Textterminal.

Beschrieben wird der Ausgangspunkt der Entwicklung, der Prozess, die wesentlichen Überlegungen, die bei der Entwicklung eine Rolle gespielt haben, und im Detail die Features der endgültigen Software.

Abstract

This work covers the design and implementation of a user-friendly application to generate and edit static robot-animations (or motions) in realtime directly on a robot via an arbitrary text-terminal.

The Thesis describes the point of origin of the development, the process, the fundamental considerations taking a role in the development and in detail the features of the resulting software.

II Inhaltsverzeichnis

1. Einführung

Obgleich in der Robotik dynamisch, aus der Situation heraus, errechnete Bewegungsabläufe an Bedeutung gewinnen, werden statisch animierte Bewegungen aufgrund ihrer Einfachheit wohl noch eine ganze Weile ein Standardansatz in der Robotik bleiben.

Professionelle Animationssoftware erfordert üblicherweise eine Software-Installation auf einem externen Gerät. Die geplanten Bewegungsabläufe werden dann dem Roboter über ein Kommunikationsprotokoll übermittelt. Besonders für Einsteiger ist es zweckmäßig, die nötige Software zur Animationserstellung nicht installieren zu müssen, wenn also der Roboter selbst die Fähigkeit besitzt, die Animationssoftware zu *hosten*.

In der vorliegenden Arbeit wird ein Ansatz entwickelt, der keine externe Installation erfordert und auch nicht im Webbrowser läuft. Trotzdem soll der Editor anwenderfreundlich und gleichermaßen geeignet für professionelle Nutzer sowie für Einsteiger, die noch keine oder wenig Erfahrung in der Robotik haben, sein.

Das vorgestellte Werkzeug wird ***Record-UI*** genannt. Der Name geht zurück auf das ursprüngliche Python-Skript, welches die Ausgangslage für die Entwicklung der *Record-UI* bildete. Dieses hieß einfach *Record-Script* (kurz: *record*), benannt nach seinem Verwendungszweck, dem *aufnehmen* (engl. *to record*) von Roboterposen zur Erstellung von Keyframeanimationen. Es handelt sich beim *Record-Script* um ein simples Komandozeilenprogramm ohne Nutzeroberfläche, weshalb die Weiterentwicklung um den Nachsatz *-UI*, für engl. *User Interface*, ergänzt wurde.

1.1. Motivation und Zielsetzung

Ziel dieser Arbeit ist es, am Beispiel der *Hamburg Bit-Bots* (siehe 1.4.3) eine Software zu entwerfen, die den besonderen Anforderungen bei der Animationserstellung im RoboCup (siehe 1.4.1) gerecht wird.

Der wichtigste Grund für die Entwicklung gegenüber der Ausgangssituation (siehe 3), ist die Notwendigkeit einer *nutzerfreundlichen* Lösung, welche sich aus dem Aufbau der Robocup-AG *Hamburg Bit-Bots* ergibt (siehe 1.4.3).

Dabei soll die entwickelte Lösung sich in die bereits vorhandene Struktur einfügen und die bekannte Arbeitsweise (siehe 1.4.4) möglichst gut unterstützen. Gleichzeitig soll die Arbeit einen nachhaltigen Mehrwert für die AG bieten und daher wartbar und gut dokumentiert sein.

1.2. Englischsprachige Fachbegriffe

Bei englischsprachigen Fachbegriffen und Wörtern oder Phrasen, die häufig aus der englischen Sprache entlehnt werden, wird im Zweifelsfalle der Prägnanz des Originalbegriffes Vorrang vor einer vollständigen deutschen Übersetzung gegeben. Durch Anpassung an den deutschen Textfluss können sich mitunter stilistische Unschönheiten ergeben, wie beispielsweise *Client-Seite* oder *gehosted*. Diese werden bewusst in Kauf genommen.

1.3. Aufbau der Arbeit

Im Anschluss an diesen einführenden Teil, zu dem im Folgenden auch eine ausführliche Erklärung des Kontext der Arbeit im RoboCup gehört, folgt im Abschnitt 2 eine Erläuterung verschiedener technischer Grundlagen, auf denen aufgebaut wird, sowie die notwendigen Begriffsverständigungen.

Im Abschnitt 3 geht es um die Ausgangslage vor Beginn der Entwicklung. Einige Funktionen und Ideen sind aus dem Programm *Record-Script* übernommen worden. Vor Beginn der eigentlichen Entwicklung werden einige Grundsatzentscheidungen (Abschnitt 4) getroffen und Vorgehensweisen (Abschnitt 5) geklärt.

Im Kernteil der Arbeit werden dann die einzelnen Entwicklungsschritte diskutiert. Die Entwicklung der ***Record-UI*** erfolgt Iterativ, *Feature-Driven*, daher entspricht dieser Aufbau logisch dem Entwicklungsprozess. Es soll nachvollziehbar werden welche Funktionalitäten erreicht wurden und was die Gründe für eine bestimmte Art der Umsetzung sind. Der Abschnitt 6 ist damit der weitaus umfangreichste.

Ähnlich aufgebaut, aber deutlich kürzer, ist der Abschnitt 7, in dem auf Funktionen eingegangen wird, welche bewusst ausgelassen wurden. Es folgt eine Ergebnisbetrachtung in Abschnitt 8 nach diversen übergeordneten Gesichtspunkten, bevor wir mit Abschnitt 9 schließlich zum Fazit kommen.

1.4. Kontext der Arbeit

Diese Arbeit entstand im Kontext des internationalen Wissenschaftswettbewerbs RoboCup (siehe 1.4.1) an dem das Team der Universität Hamburg als *Hamburg Bit-Bots* (siehe 1.4.3) seit 2012 jährlich teilnimmt.

In diesem Rahmen ist es nötig über eine Animationssoftware zu verfügen, die einerseits den professionellen Ansprüchen der erfahrenen Entwickler genügt, andererseits aber auch geeignet ist Laien einen Einstieg in die Robotik zu bieten.

1.4.1. Kurzeinführung RoboCup

Abbildung 1.1.: Logo des RoboCup
Quelle: The RoboCup Federation

Der RoboCup ist ein internationaler, wissenschaftlicher Wettbewerb mit einer Vielzahl an verschiedenen Ligen und regelmäßigen nationalen und internationalen Meisterschaften. Dabei sollen die Forschungsergebnisse durch den Wettbewerb vergleichbarer gemacht und der Wissensaustausch begünstigt werden. Zugleich soll auch allgemein die Begeisterung für das Forschungsgebiet der Robotik beim Nachwuchs und in der Gesellschaft geweckt werden.

Jede Liga hat ihren eigenen Fokus. Die drei größten Ligen sind die *@Home-League* (Servicerobotik), die *Rescue-League* (Robotik im Rettungswesen), sowie die *Soccer-League*.

Seit 1997 findet im Rahmen der Weltmeisterschaft im RoboCup auch jährlich ein Symposium statt, dessen Paper jeweils im Folgejahr als Sammlung beim Springer-Verlag unter dem Titel *RoboCup <Jahr>: Robot Soccer World Cup <Ausgabe>* erscheinen. Auf diese Sammlung sei Abseits der zu dieser Arbeit gehörenden Literatur hingewiesen.

1.4.2. Die Humanoid Kid-Size League

Es gibt viele Ligen, im Kontext dieser Arbeit ist allerdings nur die *Humanoid Kid-Size League* interessant, in welcher die *Hamburg Bit-Bots* antreten. Die *Humanoid Kid Size League* ist Teil der *Humanoid League*, welche wiederum zum Bereich *Soccer* gehört.

Thema in der gesamten *Humanoid League* ist das Voranschreiten in der humanoiden Robotik, weshalb für alle dort antretenden Robotertypen festgelegt ist, dass diese möglichst menschenähnlich aufgebaut sein müssen. Als Domäne für den Wettbewerb dient der Sport *Fußball*, welcher einen Großteil der Fähigkeiten des Menschen fordert, speziell was die Motorik anbelangt. Das Regelwerk wird dabei schrittweise an die tatsächlich geltenden FIFA-Regeln angepasst, um den Wettbewerb interessant zu halten.

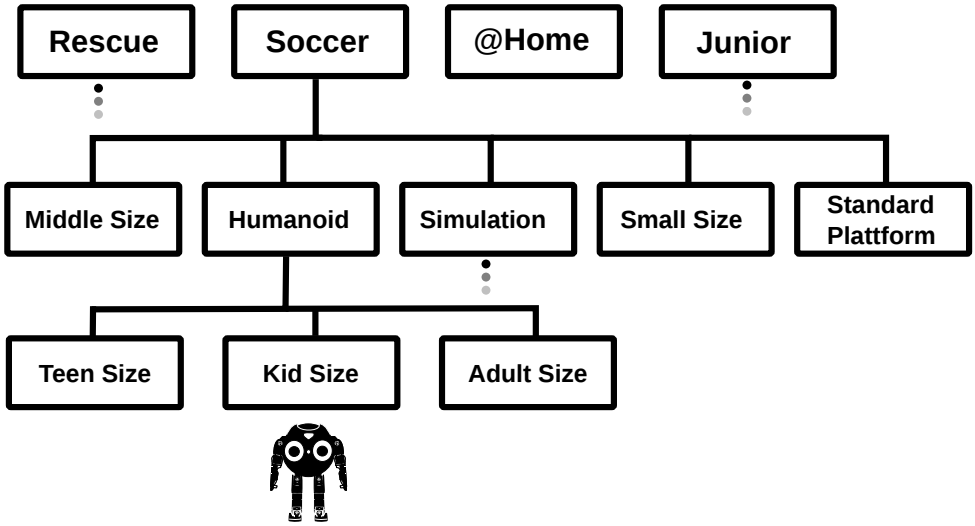


Abbildung 1.2.: Verortung der *Hamburg Bit-Bots* in den RoboCup Ligen (Ausschnitt)

Derzeit wichtigster Meilenstein der Humanoid League ist es, eine Mannschaft Menschenähnlicher Roboter bis zum Jahr 2050 in einem *fairen* Wettkampf gegen die amtierenden Weltmeister der Menschen im Fußball antreten zu lassen, also den Punkt zu erreichen, bei dem die Leistungen der Robotik in der Domäne Fußball der Spitzenleistung der Menschen entsprechen, ohne dafür Hardware einzusetzen, die nicht dem Menschen nachempfunden ist. Man kann vielleicht von einer Art Turing-Test für humanoide Roboter sprechen.

Innerhalb der Humanoid League gibt es eine Unterteilung nach Robotergröße, wobei die Kid-Size League die derzeit kleinste Baugröße von Robotern erlaubt, mit einer Höhe zwischen 40cm und 90cm [15]. Entsprechend ist es derzeit auch die beliebteste Teilliga innerhalb der Humanoid League, da hier die benötigte Hardware am einfachsten zu finanzieren ist. Die genauen Regeln der Humanoid Kid-Size League lassen sich in [15] einsehen.

1.4.3. Die RoboCup-AG *Hamburg Bit-Bots*

Bei den *Hamburg Bit-Bots* handelt es sich um eine studentische, selbstorganistierte Arbeitsgemeinschaft (AG) am Fachbereich Informatik der Universität Hamburg. Die AG hat

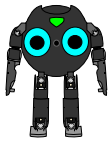


Abbildung 1.3.: Logo der *Hamburg Bit-Bots*

zur Zeit ungefähr 20 aktive Mitglieder und eine größere Anzahl Unterstützer aus der Studierendenschaft. Die Mitglieder der AG bilden die Hauptnutzergruppe für die *Record-UI*, was eine detaillierte Beschreibung der AG notwendig macht.

Die AG nimmt als *Hamburg Bit-Bots* seit 2012 am internationalen Wissenschaftswettbewerb *RoboCup* in der *Humanoid Kid-Sized-League* teil, welcher im Abschnitt 1.4.1 beschrieben wurde.

Darüber hinaus gehört es zum Selbstverständnis der AG, Wissen und Begeisterung an der Robotik an Andere weiter zu vermitteln. Zu diesem Zweck nehmen die *Hamburg Bit-Bots* regelmäßig an diversen universitären sowie öffentlichen Veranstaltungen teil, bei denen die Gäste einen Einblick in die Robotik erhalten sollen. Zum Teil gibt es gesonderte Kooperationen mit Lehrenden der Informatik, welche reguläre Lehrveranstaltung im Rahmen der AG anbieten.

Auch werden gelegentlich Projekte mit Schülern (der Mittel- und Oberstufe) ausgerichtet die einen Umfang von ungefähr einer Woche haben. Potentiell spielt dabei auch die Animationserstellung eine Rolle, da diese einen besonders guten Einstieg in die Arbeit mit einem Roboter bietet.

Alle Mitglieder der AG sind ehrenamtlich tätig und arbeiten an verschiedenen Forschungsgebieten mit Bezug zum RoboCup, normalerweise freiwillig außerhalb des Lehrplans. Viele von ihnen haben bereits die Möglichkeit genutzt, die gesammelten Erfahrungen in ihre Abschlussarbeit einzubringen.

Die AG-Mitglieder setzen sich hauptsächlich zusammen aus Bachelor- und Masterstudierenden der Informatik, sowie einiger verwandter Studiengänge. Sie befinden sich in unterschiedlichen Semestern und bringen unterschiedlichste Vorkenntnisse mit.

1.4.4. Typisches Arbeiten mit einem Roboter

Im Folgenden möchte ich die typische Arbeitsweise mit den Robotern bei den *Hamburg Bit-Bots* beschreiben, um die späteren Designentscheidungen bei der Entwicklung der ***Record-UI*** besser nachvollziehbar zu machen.

Die Mitglieder der *Hamburg Bit-Bots* entwickeln Software für den Roboter meist auf ihrem privaten Laptop oder auf gemeinschaftlich genutzten Desktoprechnern im Labor. Zum Austausch des Quellcodes wird ein Versionierungssystem verwendet.

Die Roboter besitzen eine WLAN- und eine LAN-Schnittstelle, über die sie im Betrieb mit der Netzwerkinfrastruktur verbunden sein können. Auf den Robotern läuft ein *Debian* Betriebssystem, die *Hostnames* der Roboter entsprechen ihrem eindeutigen Namen.

Soll eine bestimmte Softwareversion auf einen Roboter aufgespielt werden, so wird die Software vom AG-Mitglied auf seinem Gerät kompiliert und in einem virtuellen Robotersystem installiert. Danach wird mit Hilfe von einem *Flash-Skript*, welches das Unix-Werkzeug *rsync* verwendet, die neue Softwareversion mit dem Roboter synchronisiert. Dabei werden aus Gründen der Performanz nur die Änderungen gegenüber der vorhandenen Version übertragen.

Es werden im gleichen Zug alle kritischen Prozesse auf dem Roboter neu gestartet, damit ein neu aufgespieltes Verhalten sofort aktiv werden kann. Das Überspielen einer neuen Software auf den Roboter dauert meist nur 1-2 Minuten. Dies ist kurz genug um im Rahmen der Regeln des *RoboCup* während eines Spiels Anpassungen an der Software vorzunehmen.

Für die Arbeit an und mit dem Roboter verbinden sich die AG-Mitglieder mit ihrem Laptop über eine *SSH-Verbindung* direkt mit dem Roboter, um dort die verschiedenen Programme zu starten, zu beenden, oder die Konfiguration zu ändern. Auch notwendige Softwareänderungen an den Motoren werden soweit möglich über diesen Kanal durchgeführt. Daten, die die Roboter gesammelt haben, können ebenfalls über *SSH* auf andere Rechner übertragen werden um sie dort auszuwerten oder als Dummydaten für künftige Entwicklungen zu verwenden.

Bei vielen dieser Schritte, ist die Nutzung eines Terminals obligatorisch. Aber auch für solche Schritte für die zum Teil gute graphische Lösungen existieren wird meist auf die Kommandozeile zurückgegriffen. Zu den Vorteilen dieses Vorgehens siehe Abschnitt 2.6.

1.4.5. Eingesetzte Robotertypen

Die RoboCup-AG forscht aktiv an neuen und modifizierten Roboterplattformen. Aktuell eingesetzt werden eine Modifikation des *Darwin-OP* (siehe Abb. 1.4), einer quelloffenen Plattform, die von der Firma Robotis in Kooperation mit mehreren US-Amerikanischen Universitäten entwickelt wurde, sowie die Eigenentwicklung *Hambot* welche in [4] beschrieben wird (siehe Abb. 1.5).

Alle Roboter der *Hamburg Bit-Bots* verfügen über einen vollwertigen Computer ab 1.6 GHz Rechenleistung aufwärts mit allen gängigen Peripheriemöglichkeiten und entsprechen im Aufbau den Regeln des *RoboCup*.

Bei den eingesetzten Motoren handelt es sich ausnahmslos um Dynamixel-Motoren der Firma Robotis, welche auch den *Darwin-OP* produziert. Die Motoren besitzen eine eigene Steuerung und sind über einen gemeinsamen Datenbus ansprechbar. Dabei werden den Motoren Zielwinkel vorgegeben.

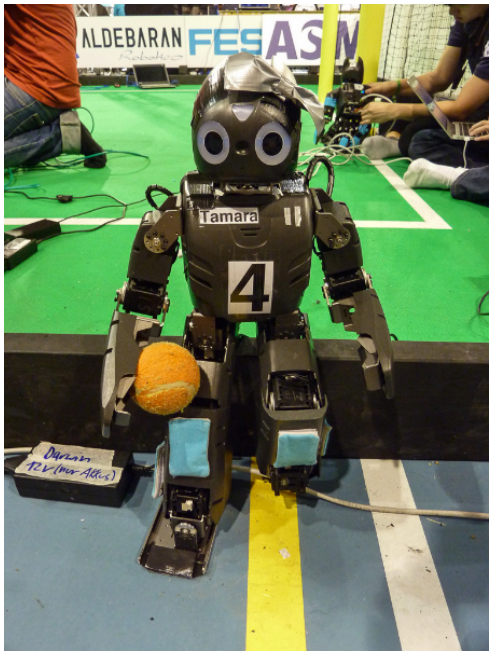


Abbildung 1.4.: Tamara, ein Darwin-OP Roboter

Jeder Motor ermöglicht genau einen Freiheitsgrad, statt von Motoren wird auch von *Gelenken* gesprochen, da in den verwendeten Robotern alle Motoren in dieser Funktion eingebaut sind[1]. Der Darwin besitzt 20 Freiheitsgrade[2], der Hambot verfügt über 24 Freiheitsgrade.

[1] Da aber auch Konstruktionen denkbar sind, bei denen die Motoren außerhalb der Gelenke angebracht sind, ist bei dieser Formulierung Vorsicht geboten.
[2] Es existiert auch eine Version mit Händen, diese besitzt 24 Freihetsgrade.

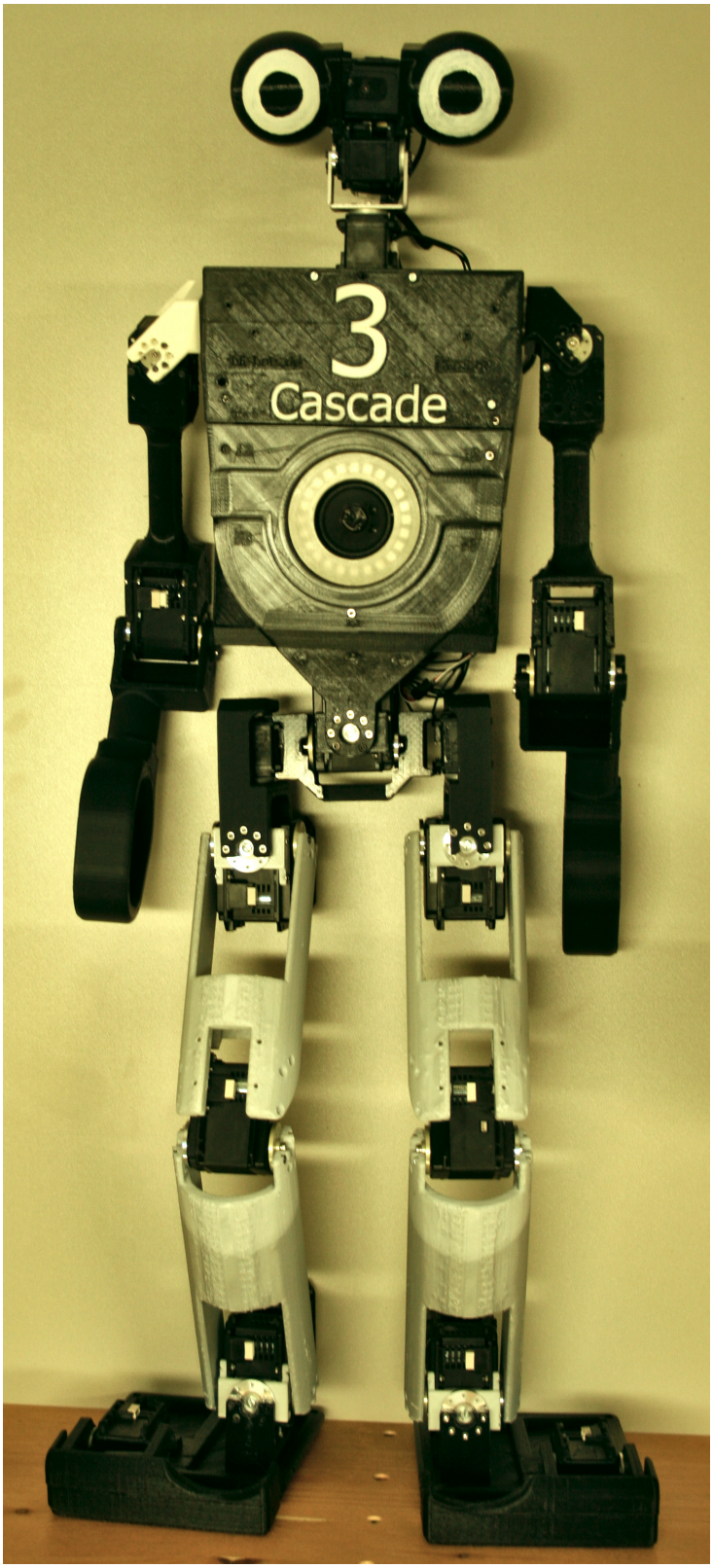


Abbildung 1.5.: Cascade, ein Roboter der Hambot-Klasse
(hier ohne Verkabelung)

2. Grundlagen

2.1. Definition Animation

Während andere Autoren nahezu durchgängig den Begriff *Motion* (wörtlich: *Bewegung*, sinngemäß: *Bewegungsablauf*) verwenden (siehe unten zur Diskussion), hat sich in der RoboCup-AG an der Uni Hamburg der weniger stark überladene Begriff der Animation für einen geplanten Bewegungsablauf etabliert. Ich werde ihn durchgängig wie folgt verwenden.

Definition 2.1.1. Eine (statische) Animation sei ein geplanter, wohldefinierter Bewegungsablauf für einen (realen oder virtuellen) Akteur.

Zu unterscheiden ist die *statische* Animation von einer *dynamischen* Animation, welche erst aus einer konkreten Situation heraus aufgrund von Sensordaten *dynamisch* entwickelt wird.

Notwendigkeit des Animationsbegriffes In der Wissenschaft besteht die Notwendigkeit für eindeutig und präzise definierte Begriffe. Durch die gesamte Literatur zieht sich bislang der Begriff der *Motion* [6], [1], [9].

Hohberg [9, S.24] versucht sich an einer Definition:

> Motions define trajectories for each joint of a Robot. These joint trajectories can consist of several joint angles, the key frames for this joint. The actual motion of a joint is created by interpolating over these key frames resulting in a function defining joint angles for the whole time of the joint's motion. Because the current pose of a robot when starting a motion is unknown, a motion determines an initialisation time that is used to bring all joints into the starting position of the motion.

Dabei wird leicht die Überladenheit des Begriffes *motion* an sich deutlich, denn dieser wird hier sowohl Synonym für *Animation*, als Begriff für die Gesamtheit des daraus resultierenden Bewegungsablaufes, sowie als Begriff für die einzelnen Motorbewegungen verwendet.

Motion ist im englischen ein allgemeiner Begriff des täglichen Lebens, was genau sich bewegt und aus welchen Gründen, ist nicht klar abgegrenzt. Auch ein ungeplantes umfallen eines Roboters kann als *motion* bezeichnet werden. Daher ist er als präziser Fachbegriff ungeeignet.

Der Begriff *Animation* (von lat. *animare* - zum leben erwecken) hingegen ist klar vom Begriff der Bewegung abgegrenzt. Er stammt aus den frühen Tagen der Tricktechnik und ist heutzutage vor allem im Bereich der Computeranimation bekannt, wo er eine nahezu identische Bedeutung einnimmt, wie auch Hohberg [9, S.3] bemerkt:

> Creating motions for robots is very similar to motion creating for animations. Thus any 3D animation tool is related to an application like the *MotionEditor*. [...] The keyframing technique which is used to generate motions for robots is an animation technique.

Darüber hinaus hat der Begriff *motion* im Kontext der *Hamburg Bit-Bots* noch eine eigene, abgeschlossene, Bedeutung. Als solche wird die Programmroutine bezeichnet, welche die Bewegungen auf dem konkreten Roboter tatsächlich im Auftrag hochleveliger Programme (wie auch der Record-UI) tatsächlich durchführt.

Statische/dynamische vs. online/offline Animationen Andernorts werden auch die Begriffe *online-* bzw. *offline-motion* synonym zu *dynamischer* bzw. *statischer* Animation verwendet, siehe Czarnetzki et al. [6]

Antonelli et al. [1, S.14] schreiben dazu jedoch:

> In general, three major strategies are adopted. The first one is about generating the motion off line, and then replaying it on the robot. The second one is about calculating the motion online, so that all the information available can be employed to produce an optimal movement. The third approach, that is placed in between the previous two, consists in calculating the motion off line and then adjusting it online depending on the data coming from the sensors, for instance to assure stability.

Antonelli et al. [1] erklärt den Unterschied zwischen *online* bzw. *offline* dadurch auf welchem Gerät die Animation entsteht.

Da in dem hier vorgestellten Ansatz die Animation üblicherweise auf dem Roboter selbst generiert wird, obwohl es sich um eine *statische* Animation handelt, würde diese alternative Wahl der Begrifflichkeiten zu Verwirrung führen.

2.2. Das Keyframeverfahren

Das Keyframeverfahren beschreibt einen statischen Bewegungsablauf über eine Reihe von *Schlüsselpositionen*[1] (Keyframes), welche zu bestimmten Zeitpunkten erreicht werden sollen.

Die Positionen zwischen den Keyframes werden interpoliert, daher ist bei einer Animation die Angabe des zu verwendenden Interpolationsalgorithmus notwendig, um die angestrebte Bewegung exakt zu beschreiben.

Definition 2.2.1. Ein Keyframe k sei im Kontext dieser Arbeit formal wie folgt definiert:

$k = \{M, p, d, i\}$

Dabei sei:

- Eine Menge M von Motor-Zielpositionen
- Eine Ablaufpause p
- Eine Ablaufdauer d (von engl. *duration*)
- Eine Interpolationsfunktion i

Die Motorzielpositonen aus M bilden dabei jeweils eine Teilmenge der Motoren eines Roboters ab. Meist handelt es sich dabei um eine *unechte* Teilmenge.

Es ist jedoch nicht zwingend notwendig, dass eine Animation, bzw. ein einzelner Keyframe als Teil einer Animation, Positionen für sämtliche Motoren eines Roboters spezifiziert. Auf diese Weise kann ein Roboter beispielsweise mit den Armen eine statische Animation ausführen, während ein Laufalgorithmus dynamisch die Beine steuert.

[1] Eine *Schlüsselposition* ist in diesem Zusammenhang die Gesamtheit der einzelnen Motorpositionen

2.3. Existierende Software

Das Problem der Animationserstellung für Roboter ist nicht neu, entsprechend existieren bereits Softwarelösungen, die im Zusammenhang mit dieser Arbeit betrachtet werden sollten.

Hohberg [9, S.3] beschreibt neben seiner eigenen Entwicklung, dem *MotionEditor* der *FUmanoids* aus Berlin, die folgenden Werkzeuge:

- *Blender* als Beispiel für 3D Animationssoftware im Allgemeinen
- *Choregraphe* der Firma *Aldebaran Robotics*[2]
- *Kouretes Motion Editor* (KME)
- Im Detail die namenlose Software, welche Ausgangspunkt für den *MotionEditor* war.

Alle von Hohberg [9] vorgestellten Werkzeuge haben eine GUI, teilweise existiert eine Visualisierung des Roboters. Alle Programme laufen dediziert auf einem Computer, die Daten werden nach der Animationserstellung auf den Roboter übertragen.

Bemerkenswert ist, dass die Software von der Hohberg [9] ausgegangen ist, ebenfalls den Roboter als Eingabegerät für die Keyframedaten verwendet, entsprechend der Ausgangssituation dieser Arbeit (siehe Kap. 3). Bei Hohberg [9] wurde diese Konzept allerdings aufgegeben, während es in dieser Arbeit verfeinert und ausgebaut wird.

Ebenfalls erwähnt sei die Software *RobovieMaker*, die nicht bei Hohberg [9] beschrieben wird. Diese Software ermöglicht es, Posen mit verschiedenen Anweisungen mit einem Graphenwerkzeug zu komplexeren Animationen und Verhaltensweisen zu kombinieren. Eine Beschreibung der Funktionsweise findet sich in [10].

2.4. Terminal

Ein Terminal dient der Ein- und Ausgabe von Textdaten in einen Computer. Ursprünglich ein Gerät, das schon vor der Einführung von Bildschirmen existierte[3], wird heute kaum eine Begriffsunterscheidung zu Terminalemulatoren getroffen, die das Verhalten in modernen Computern nachbilden.

Bekannte Terminalemulatoren unter Linux sind beispielsweise das *gnome-terminal*, oder *xterm*. Auch bei entsprechenden Zugriff auf einen entfernten Rechner z.B. mit SSH spricht man von einem (Remote-)Terminal.

[2] Dort fälschlich als *Choreographe* bezeichnet
[3] Die Ausgabe erfolgte damals auf einem Fernschreiber

Eine ausführlichere Begriffserklärung kann unnötig verwirrend sein, für das Lesen dieser Arbeit genügt es zu erklären, dass das *Terminal* lediglich die Ein- bzw. Ausgabe von Texten für eine *Kommandozeile* übernimmt.

2.5. GUI

Die GUI, englisch für *Graphical User Interface*, also graphische Benutzeroberfläche war eine Revolution der Nutzeroberflächengestaltung. Zuvor waren nahezu ausschließlich die Kommandozeilen verbreitet, die in Abschnitt 2.6 beschrieben werden.

Die GUI erlaubt eben nicht nur Text sondern auch graphische Elemente, die in der Regel mit der Maus angeklickt werden können, um mit ihnen zu interagieren. Ein Beispiel dafür sind graphische Buttons.

2.6. Kommandozeileninterface

Das *Kommandozeileninterface*, verkürzt oft *Kommandozeile*, ist eine eher historische Art der Nutzeroberfläche. Die Kommandozeile erlaubt einen Textdialog mit dem Computer unter verwendung eines *Terminals* wobei der Benutzer Kommandos über die Tastatur eingibt und ebenfalls als Text das Ergebnis vom Computer erhält. Im Englischen spricht man von einem Command Line Interface (CLI)

Bis heute haben Komandozeilen eine hohe Verbreitung und stehen auch weiterhin in jedem modernen Betriebssystem zur Steuerung zur Verfügung, auch wenn sie für den Durchschnittsbenutzer teilweise verborgen sind. Auch sind oft Kommandozeilen in GUI-Programme integriert, um die Vorteile beider Systeme zu kombinieren.

Gerade professionelle Nutzer schätzen die Eigenschaften von Kommandozeilen, sie gelten als schnell und robust. Eingaben lassen sich leicht wiederholen bzw. anpassen und vor allen Dingen kann man eine hohe Funktionalität auf engem Raum unterbringen. Bei reinen GUI Anwendungen besteht das Problem alle Funktionen auf einer begrenzten visuellen Fläche unterzubringen, was meist mit *Dropdown-Menüs* oder anderen Techniken gelöst wird, bei denen nur ein längerer Pfad zur beabsichtigten Aktion führt.

Eine Untersuchung von Tkayama and Kandogan [16] deutet darauf hin, dass Systemadministratoren als professionelle Nutzer reine Kommanodzeilenprogramme gegenüber einer GUI bevorzugen, da sie solchen Programmen ein größeres Vertrauen gegenüber bringen.

2.7. Grundsätze der Dialoggestaltung

Die DIN EN ISO 9241-110:2006 [7] beschreibt Grundlagen, wie der Dialog mit dem Nutzer gestaltet werden sollte, um eine Anwendung nutzerfreundlich zu gestalten.

Die wesentlichen Punkte die nach der EN ISO 9241-110 berücksichtigt werden sollen sind:

- Aufgabenangemessenheit
- Selbstbeschreibungsfähigkeit
- Steuerbarkeit
- Erwartungskonformität
- Lernförderlichkeit
- Individualisierbarkeit
- Fehlertoleranz

2.8. SSH

Das *Secure Shell Protocol* (SSH) ist eine offene Spezifikation für sichere, verschlüsselte und authentifizierte Verbindungen zwischen zwei Endgeräten [2]. SSH findet bereits seit den 90ern weite Verbreitung. Die technische Spezifikation von SSH liegt daher als eine ganze Reihe von *RFC* Dokumenten vor. Die wichtigsten davon sind RFC4250 bis RFC4254, wobei in letzterem Dokument die interaktive Session beschreiben wird, die später Grundlage für die ***Record-UI*** sein wird [19, S.10ff].

2.9. Python

Zur Implementation der Record-UI wird ausschließlich die Programmiersprache *Python* verwendet. Python ist eine interpretierte, plattformunabhängige Sprache, die sich durch gute Übersichtlichkeit, leichte Erlernbarkeit und eine reichhaltige Menge an quelloffenen Bibliotheken auszeichnet [11]. *Python* wird bei den *Hamburg Bit-Bots* aufgrund der Anfängerfreundlichkeit bevorzugt eingesetzt, wann immer kein hoch-performanter Code benötigt wird. In diesem Bereich wird C++ eingesetzt.

Für die Anwendungsentwicklung in Kontext dieser Arbeit ist die Geschwindigkeit von Python mehr als ausreichend.

Verwendet wird Python in der **Version 2.7.8**. Python ab Version 3.0 führt einige Neuerungen ein und ist nicht mehr kompatibel mit dem hier entwickelten Programm.

2.10. JSON

JSON steht für *Java Script Object Notation* und ist ein beliebtes Austauschformat für Strukturierte Daten, ähnlich XML. In Abgrenzung zu XML ist der Aufbau des Dokuments allerdings ein wenig vereinfacht, sodass das manuelle Lesen und Schreiben der Dokumente leichter von der Hand geht. Auch soll ein JSON-Dokument als JavaScript interpretierbar sein (siehe RFC 7159 [5]).

Bei den *Hamburg Bit-Bots* wird es als Austauschformat für die Animationsdateien verwendet, welches in Abschnitt 2.13.5 beschrieben wird. JSON wird parallel zu YAML eingesetzt

2.11. YAML

YAML ist ein rekursives Akronym für *YAML ain't markup language*. Trotzdem handelt es sich nach allgemeiner Auffassung um eine Auszeichnungssprache mit besonderem Fokus auf Lesbarkeit und einer starken Anlehnung an die Programmiersprache *Python*.

Die Abgrenzung zu JSON wird bereits in der Spezifikation von YAML beschrieben [3]:

> Both JSON and YAML aim to be human readable data interchange formats. However, JSON and YAML have different priorities. JSON's foremost design goal is simplicity and universality. Thus, JSON is trivial to generate and parse, at the cost of reduced human readability. It also uses a lowest common denominator information model, ensuring any JSON data can be easily processed by every modern programming environment.
>
> In contrast, YAML's foremost design goals are human readability and support for serializing arbitrary native data structures. Thus, YAML allows for extremely readable files, but is more complex to generate and parse. In addition, YAML ventures beyond the lowest common denominator data types, requiring more complex processing when crossing between different programming environments.

Aufgrund der besonders guten Lesbarkeit für Menschen, wird YAML inzwischen bei den *Hamburg Bit-Bots* als primäre Auszeichnungssprache verwendet, vor allem für die von Hand anzupassenden Konfigurationsdateien. Der Nachteil des aufwändigeren Parsings, welches zu Geschwindigkeitseinbußen führt, wird dadurch gemildert, dass die Dateien nur einmalig geladen und danach im Speicher vorgehalten werden.

2.12. Urwid

Bei Urwid handelt es sich um ein Python-Framework zur Gestaltung von Nutzeroberflächen für Terminalumgebungen. Wobei von einer konkreten Terminalapplikation abstrahiert wird.

Besonderes Merkmal ist ein Aufbau, der eher einer klassischen GUI-Bibliothek entspricht. Der Entwickler kann eine Vielzahl vorprogrammierter *Widgets* verwenden, die jeweils zu einer ganz besonderen Form der Darstellung und/oder Nutzereingabe geschaffen wurden.

Dabei dienen die meisten *Widgets* ihrerseits wieder als Container für untergeordnete *Widgets*, wodurch sie sich, Lego-Bausteinen gleich, relativ einfach kombinieren lassen, um komplexe Benutzeroberflächen zu bilden.

Viele der *Widgets* können sich dabei *responsiv* an die Umgebung anpassen, beispielsweise ihre Größe ändern, ihren Inhalt geeignet umbrechen oder auf eine *scrollende* Umgebung umschalten, wenn der ihnen eingeräumte Platz nicht ausreicht, um den Inhalt darzustellen.

Eine Ausführliche Beschreibung findet sich bei Ward et al. [18] auf der offiziellen Webpräsenz von Urwid.

2.13. Framework der Hamburg Bit-Bots

Die im Zuge dieser Arbeit entwickelte ***Record-UI*** ist in das Framework der *Hamburg Bit-Bots* integriert und nutzt einige der dort gebotenen Funktionalität. Es folgt ein Überblick über ausgewählte Komponenten und Eigenschaften des Frameworks.

2.13.1. Plattformabhängigkeit

Sämtliche Software wird für Linux entwickelt. Auf den Robotern der *Hamburg Bit-Bots* ist die Distribution Debian im Einsatz. Die Mitglieder der AG verwenden auf ihren Rechnern vorwiegend unixoide Betriebssysteme.

2.13.2. Der *Motionserver*

Der Motionserver ist dafür zuständig Bewegungsanweisungen von anderen Softwarekomponenten entgegenzunehmen und diese Bewegungen auf dem konkreten Roboter durchzuführen. Im Sprachgebrauch der *Hamburg Bit-Bots* hat sich eigentlich der Begriff *Motion* für den Motionserver durchgesetzt, um Verwirrung im Hinblick auf die begrifflichen Schwierigkeiten (siehe 2.1) zu vermeiden werde ich jedoch den Begriff *Motionserver* verwenden.

2.13.3. Debug-Framework

Es existiert eine Schnittstelle, die dazu gebaut wurde, während eines Wettbewerbs unidirektional Debug-Daten vom Roboter über ein Drathlosnetzwerk auf einen Zielrechner zu übertragen. Dieses wird von der Record-UI lediglich dazu genutzt sämtliche im Rahmen des Programms anfallenden Loggingdaten *zusätzlich* über diesen Kanal zu propagieren.

2.13.4. Resourcen und Config

Ein anderes Interface ist dafür zuständig Konfigurationsdaten für den Roboter zusammenzustellen, die der Übersichtlichkeit halber auf mehrere Dateien verteilt vorliegen.

Darüber hinaus besteht die Möglichkeit, die Standard-Konfiguration selektiv mit lokalen Einstellungen im Heimverzeichnis des aktuellen Nutzers zu überschreiben und Standard-Konfigurationswerte spezifisch für einen bestimmten Roboter zu laden.

Die Konfiguration der Record-UI ist in das normale Framework integriert.

Weitere Utility-Funktionen in diesem Kontext dienen dem Auffinden von Resourcendateien. Diese werden von der Record-UI genutzt, um beispielsweise Animationen, die Datei mit den Hilfetexten, sowie die Farbschemata zu laden.

2.13.5. Animations-Speicherformat

Auch existert bereits ein Speicherformat zur Spezifikation von Animationsabläufen. Die Daten werden im *JSON*-Format gespeichert.

Eine minimale Animationsdatei enthält ein Dictionary mit nur zwei Feldern: `name` vom Typ String, sowie eine Liste von Keyframe-Dictionarys mit dem Schlüssel `keyframes`.

In Abb. 2.1 und Abb. 2.2 finden sich die Datenfelder, die in einer Animationsdatei vorhanden sein können. Teilweise wird dadurch den Entwicklungen aus Abschnitt 6 vorgegriffen. Eine gekürzte Animationsspeicherdatei findet sich im Anhang unter B.3.

Schlüsselname	**Datentyp**	**Status**	**Inhalt**
name	String	Zwingend	Name der Animation
keyframes	Liste	Zwingend	Liste von Keyframe-Dictionarys
author	String	Optional	Autorenname(n)
description	String	Optional	Beschreibungstext
hostname	String	Optional	Zur Erstellung verwendetetes Gerät
last_edited	String	Optional	Zeitpunkt der letzten Änderung
version	Int	Optional	Fortlaufende Versionsnummer
default_interpolator	String	Optional	Standard-Interpolation
interpolators	Dictionary	Optional	Zuordnung Gelenke auf Interpolator

Abbildung 2.1.: Datenfelder einer Animationsspeicherdatei

Schlüsselname	**Datentyp**	**Status**	**Inhalt**
duration	Float	Zwingend	Ablaufdauer des Frames in s
pause	Float	Zwingend	Pause nach dem Frame in s
goals	Dictionary	Zwingend	Zuordnung Motoren auf Zielwerte in $^\circ$

Abbildung 2.2.: Datenfelder eine Keyframe-Dictionarys

3. Ausgangssituation

Beginn der Entwicklung bildet ein einfaches Programm zur Unterstützung der Animationserstellung (siehe Abb. 3.1). Dieses besteht im wesentlichen nur aus einer sehr simplen Form von Kommandozeile mit neun Befehlen und wird typischerweise auf dem Roboter selbst zur Ausführung gebracht.

Es ist im Wesentlichen möglich eine Animation zu laden bzw. zu speichern, die Motoren des Roboters an bzw. aus zu schalten, eine Roboterpose mit dem namensgebenden Befehl *record* aufzunehmen bzw. mit dem Befehl *revert* wieder zu verwerfen.

Fehlerbehandlung findet nur sehr begrenzt statt, bei ungültigen Eingaben erhält der Nutzer grundsätzlich die Python-Fehlermeldung als Ausgabe. Es existiert keine Autovervollständigung, einzige Benutzerhilfe ist die Ausgabe der verfügbaren Kommandos auf der Konsole, diese erfolgt als Resultat einer ungültigen Eingabe.

Größter Mangel ist allerdings, dass die aufgenommenen Posen im *record* nicht editierbar sind. Auch kann die Geschwindigkeit der Animation nicht verändert werden. Für solche Schritte muss der Nutzer nach dem Aufnehmen der Schlüsselpositionen das *record-script* verlassen und mit einem Texteditor manuell die erstellten Animationsdateien im *json-format* editieren, welche in Abschnitt 2.13.5 beschrieben wurden.

```
host@hostname:~$ record
> load
Fehler bem Ausfuehren der Funktion 'load'
Traceback (most recent call last):
  File "/home/timon/darwin/bin/record", line 132, in main
    fnmap[command](*arguments)
TypeError: load() takes exactly 2 arguments (1 given)
> help
Fehler, unbekannter Befehl.
   load, on, play, off, dump, clear, revert, record
```

Abbildung 3.1.: Das ursprüngliche Script *record* im Betrieb

3.1. Motortags

Ein wesentliches Feature des *Record-Scripts* ist die Möglichkeit eine Menge von Motoren über ihren Trivialnamen zu spezifizieren. So kann man beispielsweise den Befehl `off LArm` verwenden, um gleichzeitig alle Motoren im linken Arm lose zu stellen.

Diese sogenannten *Tags* werden direkt aus der Konfigurationsdatei `joints.yaml` entnommen. Dabei kann ein Motor beliebig viele Tags zugeordnet bekommen. Ein Auszug dieser Datei findet sich im Anhang unter B.2.

Auch Hohberg [9, S.17] hebt es als wichtig hervor die Trivialnamen der Motoren aufzuführen. Er schreibt:

> Motors should not only be named with IDs but also with names describing the equivalent human joint.

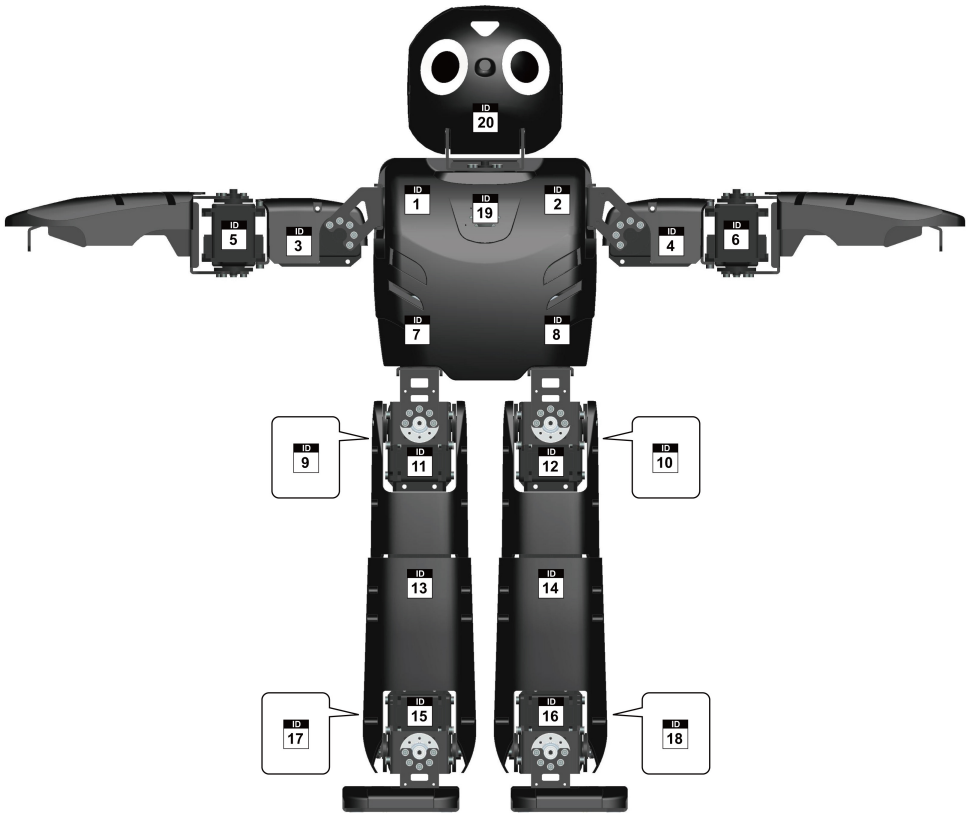


Abbildung 3.2.: MotorIDs des Darwin-OP, die Zuordnung ist nicht intuitiv
Bildquelle: Robotis (siehe [14])

4. Grundüberlegungen

4.1. Motivation: Nutzerfreundlichkeit

Die Notwendigkeit einen besonderen Fokus auf die Nutzerfreundlichkeit zu legen, ergibt sich aus der in Abschnitt 1.4.3 beschriebenen Nutzergruppe.

Ein Einstieg in die RoboCup-AG soll für neue Mitglieder möglichst leicht sein. Aufgrund der ehrenamtlichen Natur wäre eine professionelle Schulung durch eine Vollzeitkraft im Hinblick auf die verwendete Software nicht praktikabel. Es ist daher ein notwendiges Kriterium für die ***Record-UI*** , dass die Bedienung einfach zu erlernen ist, wo es möglich ist sogar intuitiv. Darüber hinaus ist es in diesem Fall besonders wichtig, dass ein Neuling frühzeitig ein Erfolgserlebnis bei der Nutzung der ***Record-UI*** hat, da er keine wirtschaftliche Motivation hat, mit der Software zu arbeiten. Im Idealfall hat er bereits bei der Benutzung Spaß, zumindest soll er aber nicht von ihr frustriert werden[1].

Neben neuen, regulären, AG-Mitgliedern gibt es wie erwähnt auch Schülerprojekte und ähnliche Veranstaltungen, bei denen Anfängern die Robotik näher gebracht werden soll. Für diese ist es besonders interessant einen Roboter selbstständig in Bewegung bringen zu können.

Es darf bei beidem aber nicht außer Acht gelassen werden, dass die Anwendung zum produktiven Einsatz in der Robotikforschung bestimmt ist und z.T. der Erfolg beim RoboCup von der ***Record-UI*** abhängen wird. Sie muss daher gleichermaßen *mächtig* genug für den professionellen Einsatz sein. Es ist damit keine Lösung nur einen minimalen Funktionsumfang zu implementieren, die ***Record-UI*** ist nicht als Lernsoftware gedacht.

Die Grundlagen der Dialoggestaltung nach DIN EN ISO 9241-110:2006 [7] (siehe 2.7) sollen für die Nutzerfreundlichkeit als Leitlinien hinzugezogen werden. Eine Zertifizierung der Software wird dabei allerdings nicht angestrebt.

[1] Kein Nutzer sollte je von einer Software frustriert werden, hier ist es aber von besonderer Wichtigkeit

4.2. Fortführung als Terminalprogramm

Eine wichtige Frage, die vermutlich beim Lesen des Titels entsteht, ist: *Warum überhaupt die Ausführung als Terminalprogramm?*

Die Wiederverwendung des bereits vorhandenen Codes ist kein besonders starkes Argument, eine UI ist nicht vorhanden, die Geschäftslogik kann auch bei anderen Ansätzen verwendet werden.

Ein wichtiger Grund für die Entscheidung, ist die enorme Flexibilität, die eine Terminalanwendung im Hinblick auf das verwendete Gerät bietet. Terminals lassen sich in Verbindung mit fast jedem System nutzen, dank dem SSH-Protokoll (siehe 2.8) kann man sich von beliebigen Geräten aus mit der auf dem Roboter laufenden Software verbinden. Genauso ist es aber möglich eine Tastatur und einen Monitor mit dem Roboter zu verbinden und die ***Record-UI*** direkt zu nutzen, ohne dass dieser graphische Oberflächen unterstützen muss[2].

Durch die Ausführung der Software auf dem Roboter, kann dieser direkt als *Eingabegerät* verwendet werden. Das wäre allerdings auch über eine Client-Server Architektur erreichbar. Der Nachteil bei einer solchen ist, dass dabei zum einen zwei Softwarekomponenten gefplegt werden müssten, zum anderen eine weitere Installation auf der Client-Seite nötig ist.

Schließlich muss auch immer sichergestellt sein, dass die Version des Clients mit der Serverversion kompatibel ist, was erheblichen Aufwand mit sich bringt.

Beim Ansatz der von Hohberg [9] beschrieben wird, handelt es sich um eine solche Client-Server-Architektur, allerdings integriert in ein bereits vorhandenes Framework, weshalb das Vorgehen dort nur konsequent ist.

Bei den *Hamburg Bit-Bots* hingegen wird wie in 1.4.4 beschrieben meist direkt im Terminal gearbeitet. Eine Fortführung der ***Record-UI*** als Terminalprogramm sorgt also dafür, dass sie sich in die bereits vorhandene Softwarelandschaft einfügt.

An dieser Stelle sei auch noch ein dritter, gewissermaßen hybrider, Ansatz erwähnt, den ich bei einem anderen Team des RoboCup beobachtet habe, und zu dem es bislang leider keine Literatur gibt.

[2] Alle Roboter der *Hamburg Bit-Bots* wären theoretisch in der Lage auch eine grafische Oberfläche zu nutzen, aus Gründen der Resourcenersparnis wurde das Betriebssystem aber ohne Grafikfunktionalität installiert

Dort wird als Client ein normaler Webbrowser eingesetzt, ob der Roboter direkt als Webserver fungiert ist mir unbekannt. Wenn dies der Fall ist, wird dort der Vorteil einer mächtigen graphischen Oberfläche ohne den Nachteil einer Externinstallation genutzt.

5. Vorgehen

Bei der Entwicklung der ***Record-UI*** wird Feature-Getrieben vorgegangen. Ausgehend von der Basisversion, dem alten *Record-Script* werden möglichst kleinschrittig Verbesserungspotentiale identifiziert und jeweils eine Lösung Entwickelt, um diese zu nutzen.

Grundlage sind dabei die eigenen Erfahrungen der Arbeit in der Robocup-AG, sowie Feedback und Wünsche der anderen AG-Mitglieder. Eine Spezifikation existiert nicht. Bei den Entwicklungsschritten in Abschnitt 6 wird jeweils die Motivation und die Lösung aus Nutzersicht diskutiert. Nur dort wo es interessant erscheint wird auch näher auf die technische Umsetzung eingegangen.

In den folgenden Abschnitten wird näher auf Themen eingegangen, die den Entwicklungsprozess in seinen Grundzügen prägen.

5.1. Softwaretests

Um ein Mindestmaß an Softwarequalität zu gewährleisten, werden die wichtigsten Komponenten der ***Record-UI*** mit Hilfe von Unittests automatisiert getestet, insoweit der Aufwand der Testentwicklung in einem angemessenen Verhältnis zum Nutzen steht.

Eine testgetriebene Entwicklung findet nicht statt, da die Spezifikation demselben Geist entsprungen ist wie die Implementierung. Ohne eine von außen gegebene Spezifikation, ist der Nutzen einer testgetriebenen Entwicklung gering. Die meisten Tests werden also nach der Implementierung eines *Features* geschrieben, um dieses mit verschiedenen Eingaben zur Ausführung zu bringen und zu testen, ob das gezeigte Verhalten den Erwartungen entspricht.

Bei später beobachtetem Fehlverhalten[1] der ***Record-UI*** wird abweichend grundsätzlich nach dem *Test-First* Ansatz die Fehlerbehebung vorgenommen, um sicherzustellen, dass der beobachtete Fehler tatsächlich beseitigt wurde und um einen Regressionstest für die Zukunft zu erhalten.

[1] Ein Fehlverhalten im Kontext der Entwicklung der ***Record-UI*** entspricht hier dem Abweichen von den Erwartungen des Autors.

Es wird kein besonderer Wert darauf gelegt die einzelnen Unittests unabhängig von einander zu gestalten. Es kann also der Fall sein, dass Tests fehlschlagen, weil eine andere Komponente als die gerade getestete, ein Fehlverhalten zeigt. Dies kann die Ursachensuche behindern, verursacht aber einen geringereren Aufwand bei der Testentwicklung. Darüber hinaus kann man eine quantitative Abschätzung treffen, wie schwerwiegend bzw. weitreichend ein eingeführter Fehler ist, indem man beobachtet, wie viele Testfälle davon betroffen sind.

Testbarkeit Es wird bei der Entwicklung der ***Record-UI*** Wert darauf gelegt, die einzelnen Komponenten möglichst testbar zu gestalten. Daher sollen alle Methoden, die eigentlich keinen Rückgabewert benötigen, grundsätzlich einen *Boolean* zurück geben, welcher darüber Auskunft gibt, ob die in der Methode durchgeführte Aktion als *Erfolg* oder *Misserfolg* gewertet werden soll. So genügt es für einen einfachen Test eine Methode auszuführen und in Kenntnis der Ausgangslage zu prüfen, ob der Rückgabewert dem erwarteten Erfolgsergebnis entspricht.

Ein Beispiel für einen Testcase findet sich im Anhang (B.1).

Weitere Motivation Neben der erwähnten Softwarequalität spielt es eine große Rolle, dass die ***Record-UI*** auch nach Abschluss dieser Abschlussarbeit produktiv genutzt, und auch weiter entwickelt werden soll. Softwaretests sind gewissermaßen ein Teil der Dokumentation, welcher automatisiert überprüfbar ist.

5.2. Wiederverwertbarkeit des Codes

Es ist akzeptierter Standard in der Softwareentwicklung die Programmierung der Nutzeroberfläche von der Entwicklung der Geschäftslogik zu trennen. Dies soll es in der Zukunft einfacher machen, Änderungen an einer der beiden Komponenten vorzunehmen – sie im Idealfall sogar völlig austauschbar machen.

Hohberg [9][S.10, S.30] beschreibt in seiner Arbeit das *MVA-Designpattern*, welches er benutzt, um dieses Ziel zu erreichen. In dieser Arbeit wird weitestgehend auf eine Vermittlungsinstanz, den *Controller* bwz. *Adaptor*, verzichtet.

Die Trennung erfolgt stattdessen in zwei logische Komponenten, das *record* und die *UI*. Jede Komponente hält die Repräsentation, die sie benötigt.

Durch die Verwendung von *Python* wird die Auftrennung von Softwarekomponenten zusätzlich etwas vereinfacht, da es keine starke Typisierung gibt. Python basiert auf dem

Prinzip des sogenannten *Duck-Typing*, nach dem Motto: *If it quacks like a duck, it is a duck!*

Eine Komponente lässt sich daher problemlos durch eine andere austauschen, solange diese die bislang verwendeten Methoden anbietet - und sich auch die zurückgegebenen Objekte wie bisher verwenden lassen. Es ist daher im Vergleich zu *Java* für die Substituenten nicht notwendig (expliztit) ein gemeinsames Interface zu implementieren.

5.3. Programmierstandards

Die Arbeit erfolgt im Rahmen der *Hamburg Bit-Bots* , daher werden auch alle Konventionen übernommen, die dort üblich sind.

Was Python angeht sind grundsätzliche Stilfragen durch den Python-Standard PEP8 (siehe [17]) geregelt, der sehr weite Verbreitung hat und auch bei den *Hamburg Bit-Bots* berücksichtigt wird.

Weiteres Kriterium ist englischsprachiger Quellcode inklusive Quellcodedokumentation. Die Ergebnisse der Arbeit sollen im internationalen Forschungsaustausch verwendet werden. Auch gibt es am Fachbereich Informatik der *Universität Hamburg* einen internationalen Masterstudiengang, dessen Teilnehmer oft kein Deutsch sprechen, was einen zusätzlichen Grund darstellt.

5.4. Dokumentation

Es gibt zwei Arten von Dokumentation, die für diese Arbeit wichtig erscheinen. Zum einen die Dokumentation für die Entwickler und zum anderen die Dokumentation für die Benutzer. Es ist anzunehmen, dass alle künftigen Entwickler gleichzeitig Benutzer der Software sein werden, es werden aber nicht alle Nutzer Entwickler sein.

5.4.1. Entwicklerdokumentation

Die Entwicklerdokumentation soll quellcodenah erfolgen. Dabei gibt es in Python die sogenannten *Docstrings*, welche vor allen Dingen zur Beschreibung von Klassen und Methoden herangezogen werden[2] (siehe Abb. 5.1).

Docstrings sind in Python stark integriert, so kann man an einem Python-Objekt – auch Methoden sind Objekte – grundsätzlich direkt den Docstring aufrufen. Das zeigt wie eng die Programmierung in Python mit der Dokumentation verknüpft ist. Die Docstrings geben speziell bei Methoden wichtige Hinweise welchen Zweck diese erfüllen sollen, welche

[2]Funktionen, also klassenlose Methoden werden in diesem Abschnitt nicht von Methoden unterschieden

```
class Beispielklasse(object):
  """ Erklärt Docstrings
  """

  def methode(zahl, string):
      """ Eine Beispielmethode

      :param zahl: Ein Zahlenwert
      :param string: Ein String
      :returns: den String
      """
      return string
```

Abbildung 5.1.: Beispielhaftes Python-Dokument, das ein paar Docstrings zeigt

Parameter erwartet werden, welche Rückgabewerte zu erwarten sind und welche *Exceptions* möglich sind.

Eine schlechte Dokumentation wäre in Python aufgrund der schwachen Typisierung besonders problematisch, da der Datentyp von Variablen erst zur Laufzeit bestimmbar ist.

Neben den speziellen Docstrings ist es natürlich auch in Python üblich, Anmerkungen in den Quellcode einzubringen, die künftige Entwickler auf besonderheiten des Codes aufmerksam machen sollen. Zur Entwicklerdokumentation gehören daneben auch sprechende Variablen- und Methodennamen.

Online-Entwicklerdokumentation

Zusätzlich zur Dokumentation im Quellcode existiert bei den *Hamburg Bit-Bots* eine generierte Onlinedokumentation in der sowohl Entwicklerdokumentation als auch Benutzerdokumentation gesammelt werden. Für die Onlinedokumentation wird die Software *Sphinx* eingesetzt, die es unter anderem erlaubt, die Quellcodedokumentation aus Python automatisch einzubinden.

Da es sehr üblich ist, solche Techniken zu Dokumentationsgenerierung zu verwenden, werden Docstrings in Python üblicherweise im *Restructured-Text-Markup* (RST) verfasst. So natürlich auch in dieser Arbeit.

Mit dem System *Sphinx* kann auch beliebig komplexe Dokumentation außerhalb des Quellcodes ergänzt werden, wenn z.B. komplexere Zusammenhänge geschildert werden sollen, die nicht irgendwo direkt zum Quellcode passen.

5.4.2. Benutzerdokumentation

Wie zuvor kurz angeschnitten wird auch die Benutzerdokumentation der *Hamburg Bit-Bots* mit Sphinx in ein Onlinedokumentationssystem eingefügt. Daher ist auch das Erstellen von entsprechender Dokumentation Bestandteil dieser Arbeit.

Die Benutzerdokumentation wird innerhalb des Onlinesystems von der Entwicklerdokumentation getrennt.

5.5. Logging mechanismus

Bei der Entwicklung von Programmen größeren Umfangs sollte man sich Gedanken über den Nachrichtenfluss machen. Dies betrifft sowohl die Meldungen an den Benutzer, als auch (Debug-)Meldungen an die Entwickler. In der *RoboCup-AG* ist der Übergang zwischen Entwickler und Benutzer fließend. Bereits beim Entwurf der Anwendung wird daher darauf Wert gelegt, dass Debuginformationen direkt in die Nutzerausgabe umgeleitet werden können.

Zu diesem Zweck soll auch die Nutzerausgabe direkt auf dem Standard-Logging-System von Python aufgesetzt werden. Bei diesem System können beliebige Textmeldungen, mit einem *Debuglevel* versehen, propagiert werden. Der Debuglevel ist ein mit der Nachricht assoziierter Integer-Wert, der die vermutete Wichtigkeit der Nachricht repräsentiert. Typischerweise wird zwischen den Leveln `DEBUG, INFO, WARNING` und `ERROR` unterschieden. Die genaue Definition dieser Level variiert, für diese Arbeit sollen diese Level die folgende Bedeutung tragen:

Nachrichten des Levels `DEBUG` dienen dazu einem Entwickler Informationen über den (regelhaften) internen Ablauf des Programms zu vermitteln, um diesen bei einer Fehlersuche zu unterstützen.

Nachrichten des Levels `INFO` sind solche Meldungen, die den Benutzer über den regelhaften Ablauf eines Vorgangs informieren.

Nachrichten des Levels `WARNING` informieren über den nicht-regelhaften Ablauf eines Vorgangs, welcher aber als nicht kritisch erachtet wird, z.B weil ein Fehler automatisch behandelt werden kann.

Nachrichten des Levels `ERROR` benachrichtigen über einen kritischen Fehler, der möglicherweise den weiteren regelhaften Ablauf des Programms gefährdet.

In Abb. 5.2 ist der Nachrichtenfluss für die ***Record-UI*** dargestellt. Wie dort zu sehen ist, soll die Möglichkeit bestehen, Debug-Informationen optional in eine Datei zu schreiben.

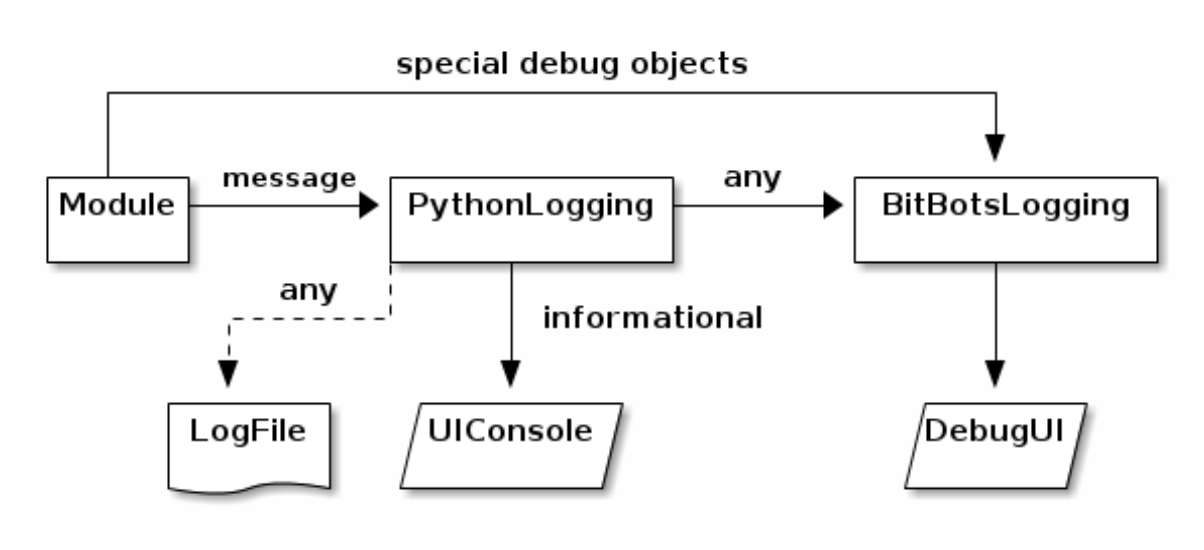


Abbildung 5.2.: Übersicht über den Nachrichtenfluss der ***Record-UI***

Nachrichten des Levels `INFO` oder wichtiger werden in der ***Record-UI*** direkt für den Nutzer ausgegeben. Alle Daten werden standardmäßig auch an das bereits vorhandene Debug-Framework der *Hamburg Bit-Bots* gesendet[3].

[3]Die Debug-Verbindung ist unverschlüsselt. Siehe 8.5.2 zu den Sicherheitsimplikationen

6. Entwicklungsschritte

6.1. Änderungsmöglichkeit für die Animation

Der erste und wichtigste Schritt bei der Entwicklung der Record-UI besteht darin, die aufgenommenen Daten bearbeitbar zu machen, ohne dafür einen externen Editor zu verwenden. Dabei soll ein Überblick über die derzeit bearbeitete Animation geschaffen werden.

6.1.1. Anzeige der Keyframes

Die Kernfrage ist, wie überhaupt eine Menge von (mindestens) 20 Motoren, plus Werte für Ablaufdauer und Pause, in einem einfachen textbasierten Terminal unter den dort gegebenen Limitierungen repräsentiert werden kann. Da die Menge der Keyframes einer Animation nicht beschränkt ist, ist es nötig, dass der Benutzer die Möglichkeit hat, die Übersicht zu *scrollen* um ihm möglicherweise unzugängliche Informationen zu betrachten. Spätestens mit dieser Überlegung steht fest, dass ein professionelles Framework zur Gestaltung der Oberfläche benötigt wird, welches von der Ausgabe des Textes zur Gestaltung der Pseudo-Graphischen Ansicht von Nöten ist. Ich habe mich hierzu für das Python-Framework *Urwid* entschieden (siehe 2.12).

Eine kompakte und übersichtliche Möglichkeit der Darstellung eines einzelnen Keyframes ist eine Listenanordnung, bei der jeder Motor ein einzelnes Formularfeld zum Anpassen der Werte besitzt. Dieses kann der Nutzer auswählen und bearbeiten, dabei muss klar ersichtlich sein zu welchem Motor ein Feld gehört und auch zu welchem Keyframe. Rahmenlinien sind als Trenner der Komponenten von Nöten, sie funktionieren auch in monochromen Terminals noch als mittel der optischen Trennung.

Bei Terminals mit Farbunterstützung können die speziell für die Dauer bzw. Pause eines Keyframes nötigen Felder auch farblich und somit zügig von den normalen Zielwinkelfeldern unterschieden werden.

Es soll vermieden werden, dass der Benutzer auf mehr als einer Achse *scrollen* muss, um die Übersicht bestmöglich zu wahren. Daher werden alle Elemente am Zeilenende des

Terminals umgebrochen. Glücklicherweise unterstützt das *Urwid* Framework einen solchen Umbruch bereits, und tut dies in einer Weise, dass der Zusammenhang der Datenfelder als graphische Einheit erhalten bleibt.

In der Realität ergibt sich durch den automatischen Umbruch der Liste mit Datenfeldern optisch eine tabellarische Form für den Benutzer, wie in Abb. 6.1 sowie 6.2 zu sehen.

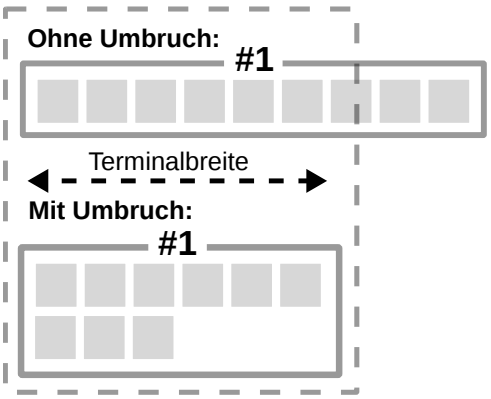


Abbildung 6.1.: Zeilenumbruch um horizontales Scrolling zu vermeiden

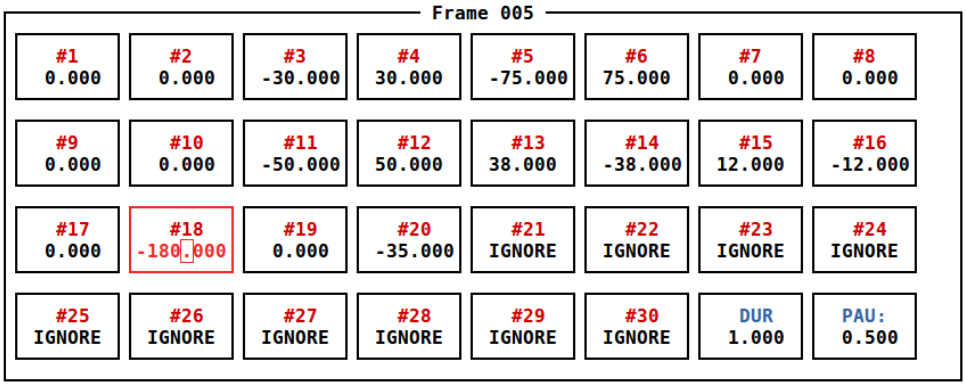


Abbildung 6.2.: Ein einzelner Keyframe der fertigen ***Record-UI***

Anordnung der Keyframes

Analog zur Anordnung der Datenfelder innerhalb der Keyframes, wird die logische Anordnung der Keyframes als Liste abgebildet. Dabei erhält auch jedes Keyframeelement eine Rahmenlinie und einen eindeutigen Titel, welcher der Position des Keyframes in der Animation entspricht.

Die Anordnung der Keyframes in der Ansicht erfolgt vertikal, während die Anordnung der Datenfelder innerhalb eines Keyframes primär horizontal erfolgt.

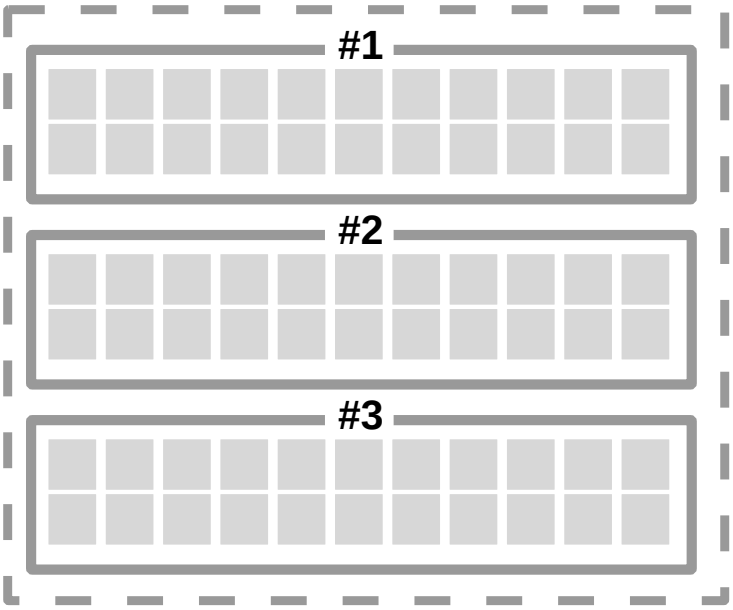


Abbildung 6.3.: Konzept der Keyframe Anordnung

6.1.2. Platzierung der Kommandozeile

Als nächstes soll die Funktionalität des alten *record-scripts* untergebracht werden. Dazu wird im Bereich unter der Keyframeliste ein Areal für die Text Ein- und Ausgaben vorgesehen, dort wo man ihn als Nutzer anderer CLI-Programme und Terminalemulatoren erwartet.

Ein Problem ergibt sich allerdings daraus, dass in einem Terminal nur eine begrenzte Menge an Zeilen Platz findet. Bei einer ungünstigen Größe der Schrift und des Terminalfensters, kann ein einzelner Keyframe bereits die Hälfte des verfügbaren Sichtbereiches einnehmen. Wenn nun das Terminal die komplette untere Hälfte des Sichtbereiches einnimmt, schmälert dies den Sichtbereich für den Nutzer, und macht verstärktes *Scrolling* in der Keyframeliste erforderlich. Das Terminal sollte also möglichst klein sein, um Platz für die Keyframeliste zu lassen.

Auf der anderen Seite bietet das alte *record-script* einen Verlauf der bislang eingegebenen Kommandos. Auch wenn es dort nicht möglich ist alte Kommandos zu wiederholen, ist es für den Nutzer hilfreich eine Historie seiner Aktionen vor Augen zu haben. Je weniger Platz die Kommandozeilenansicht zur Verfügung hat, um so kürzer fällt der sichtbare Bereich der Historie aus und um so mehr *Scrolling* wird hier erforderlich. Das Terminal sollte also möglichst groß sein, um Platz für die Kommandohistorie zu bieten.

Eine Lösung für dieses reziproke Problem findet sich darin, die Ansicht dynamisch zu verändern, je nachdem ob der Fokus des Nutzers gerade auf der Kommandozeile oder der Keyframeansicht liegt. So wird die Kommandozeile expandiert, wenn der Benutzer beginnt

dort eine Eingabe zu machen, und wieder geschrumpft sobald er fortfährt Keyframewerte zu verändern. Dieses Konzept ist nicht neu und findet auch in anderen Programmen anwendung, die ein CLI verwenden.

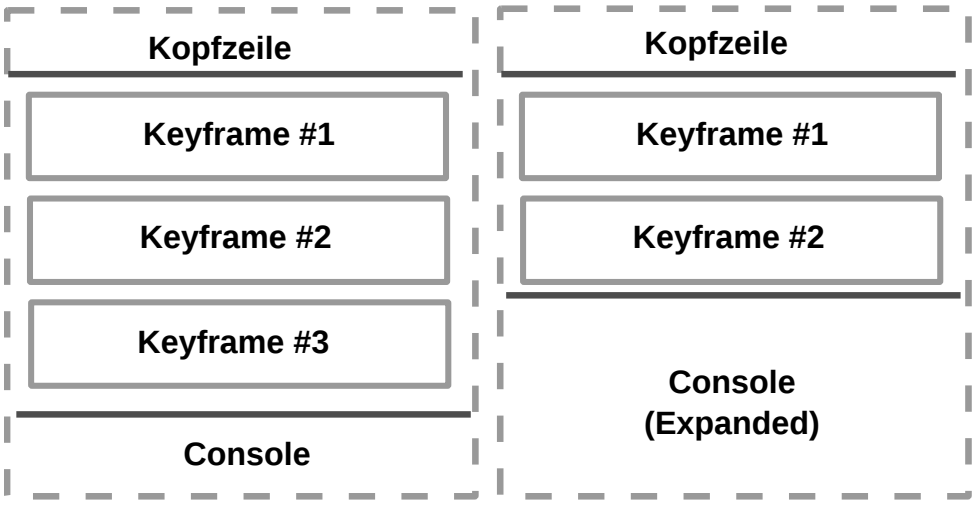


Abbildung 6.4.: Konzept der Konsolenexpansion

6.1.3. Hervorhebung der Auswahl

Es ist notwendig, dass der Nutzer erkennt wo er eine Eingabe tätigt. Dies wird wie folgt umgesetzt: Selektierte Elemente werden bei entsprechenden Fähigkeiten des Terminals farblich hervorgehoben. Auch ist jederzeit ein *Cursor* vorhanden, welcher die aktuelle Eingabeposition darstellt. Die Darstellung des Cursors ist Angelegenheit des verwendeten Terminals. Viele Terminals stellen den *Cursor* blinkend dar, was besonders bei monochromen Terminals ein wichtiger optischer Hinweis auf das aktuell selektierte Element ist.

Es sollte dem Nutzer grundsätzlich nicht möglich sein ein Element zu fokussieren, welches ihm keine Eingabe erlaubt. Dies würde ihm keinen Mehrwert bringen, aber bei der Navigation stören, welche in Abschnitt 6.1.4 beschrieben wird.

6.1.4. Steuerungskonzept

Zuletzt müssen noch die Nutzereingabemöglichkeiten implementiert werden.

Mit einer vorhandenen Maus, können sämtliche selektierbaren Objekte ausgewählt werden, die im Sichtbereich des Terminals liegen. Dies ist Teil der Standardfunktionalität von *Urwid* Es ist jedoch technisch nicht möglich, die Maus zum *Scrollen* der Ansicht zu verwenden, insbesondere kann das Mausrad nicht von der Anwendung genutzt werden, da diese Eingaben nicht durch ein Terminal propagiert werden.

Das primäre Mittel der Eingabe soll die Tastatur sein. Tasteneingaben sind gegenüber der Maus sehr präzise und schnell, wenn der Benutzer weiß welche Taste zu drücken ist.

Sowohl die Kommandozeile als auch die Eingabefelder erfordern normale Texteingabe, diese sollte für einen Benutzer kein Problem darstellen. Wichtiger ist die Navigation durch die Eingabefelder, welche im Folgenden diskutiert wird.

Triviale Bewegungen

Ein ungeschulter Benutzer wird mindestens die trivialen Bewegungen *hoch, runter, links, rechts* erwarten. Diese werden natürlich über die Pfeiltasten auf der Tastatur realisiet. So ist es möglich jeweils zum nächsten aktivierbaren Element in der angegebenen Richtung zu *springen*, dabei *scrollt* die Ansicht bei Bedarf mit. Neben den trivialen Bewegungen *hoch, runter, links, rechts* gibt es weitere weit verbreitete Bewegungsmuster.

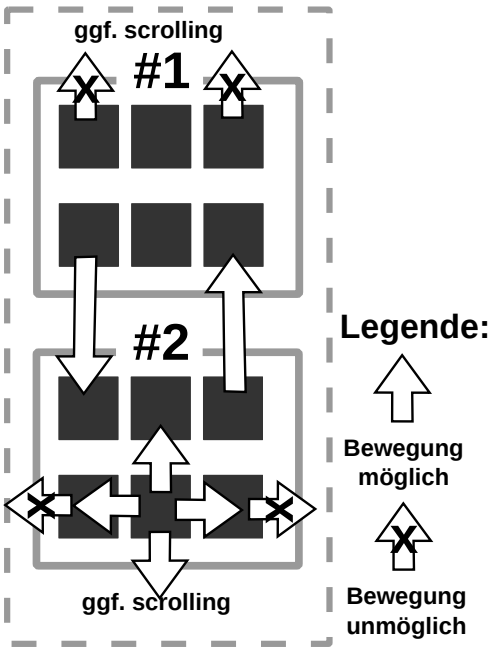


Abbildung 6.5.: Ausgewählte Navigationsmöglichkeiten mit den Pfeiltasten

Tabulatorsprung

Meist ist der Sprung zum nächsten selektierbaren Element in Formularen über den *Tabulator* aufrufbar. Dies wird auch in der ***Record-UI*** umgesetzt. Dabei soll allerdings nicht die komplette Oberfläche des Programms als ein Formular gelten. Die typische Nutzung soll Keyframeweise und das Durchwecheseln der aktiven Elemente innerhalb eines Keyframes zyklisch erfolgen.

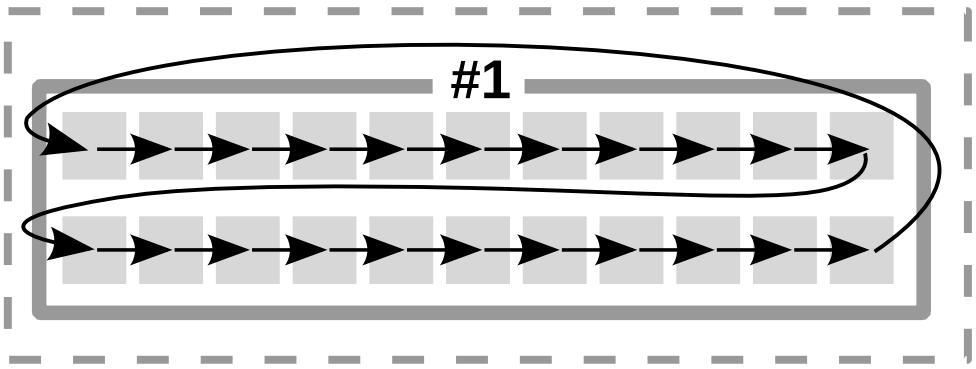


Abbildung 6.6.: Navigation mit dem Tabulator

VIM-Artige Navigation

Die Navigation mit den Pfeiltasten ist intuitiv, aber nicht ergonomisch. Beim Tippen im Zehnfingersystem hat man als Benutzer in Ruhestellung die Finger auf der sogenannten *Homerow* liegen.

Um zu vermeiden, dass die Benutzer immer wieder die Schreibstellung verlassen müssen, bietet der weit verbreitete Terminaleditor *vim* ein besonderes Navigationskonzept. Zusätzlich zu den Pfeiltasten werden dort die tasten `h` (links), `l` (rechts), `k` (hoch) und `j` (runter) verwendet, auf denen die rechte Hand in Ruhestellung bei den gängigsten Tastaturlayouts liegt. Diese Steuerung soll auch in der ***Record-UI*** Anwendung finden.

Da zumindest einige Mitglieder der *Hamburg Bit-Bots* mit dem Steuerungskonzept von *vim* vertraut sind, wurde versucht weitere nicht-triviale Navigationsmöglichkeiten an dieses anzulehnen. Beispielsweise das Springen zum Anfang bzw. zum Ende der Keyframeliste mit `g` bzw. `G`.

6.2. Scrollindikatoren

Mit den Änderungen, die in 6.1 beschrieben wurden, erhalten wir bereits eine nutzbare Version, bei der der Benutzer eine aufgenommene Animation bearbeiten kann. Es wird allerdings bei der Verwendung ein Problem offensichtlich, dass von der *Urwid*-Bibliothek nicht zufriedenstellend berücksichtigt wurde.

Bei einer langen Liste von Keyframes finden nicht alle Keyframes Platz in der Höhe des Terminals. In diesem Falle ermöglicht *Urwid* zufriedenstellend das Scrollen um alle Keyframes betrachten und editieren zu können. Wie aber erkennt der Benutzer, dass weitere Keyframes vorhanden sind, die außerhalb seines Sichtbereiches liegen? Wie erkennt er, dass er am Ende der Liste mit Keyframes angelangt ist? *Urwid* bietet von Haus aus keine Anzeige dafür.

Als Lösung für dieses Problem führen wir *Scrollindikatoren* ein, welche am oberen bzw. unteren Ende der Ansicht platziert werden, wenn ein *Scrolling* in diese Richtung möglich ist. Diese Indikatoren bestehen aus Zeilen des Characters $\triangle$ bzw. $\triangledown$, um anzuzeigen dass ein Bildlauf in diese Richtung möglich ist.

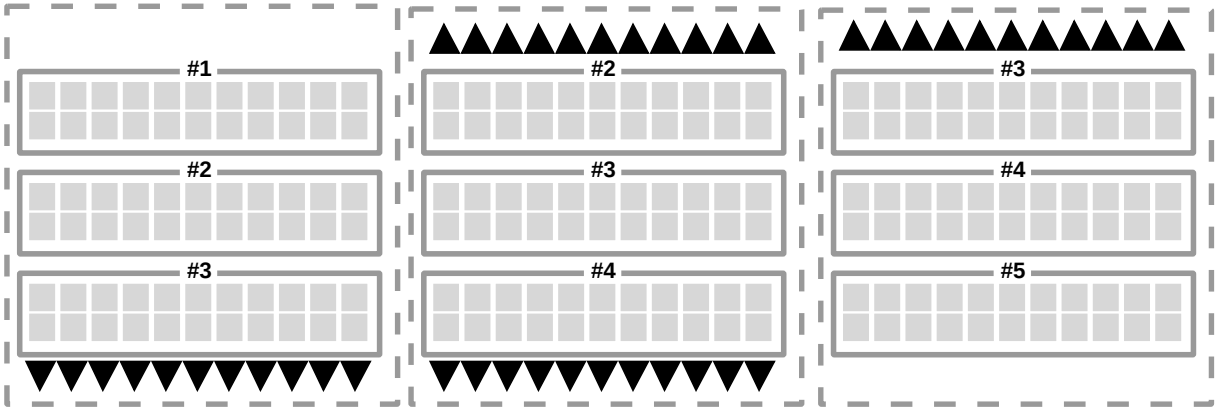


Abbildung 6.7.: Funktionsweise der Scrollindikatoren

6.3. Autovervollständigung

Der nächste Schritt soll die Unterstützung des Benutzers bei seinen Eingaben sein. Bislang muss ein Nutzer seine Eingabe vollständig tätigen. Ein wesentliches Feature von CLI-Anwendungen ist die automatische Vervollständigung der Eingabe, sofern möglich. Dies reduziert auch die Wahrscheinlichkeit von Tippfehlern.

6.3.1. Algorithmus zur Autovervollständiung

Zunächst tätigt der Nutzer eine (unvollständige) Texteingabe, bevor er über ein Tastenkürzel die Autovervollständigung startet. Sollte keine Vervollständigung existieren, dann wird eine entsprechende Information ausgegeben damit er weiß, dass der Vorgang abgeschlossen wurde und merkt, dass er bei der Eingabe einen Fehler gemacht hat.

Existiert genau eine Vervollstädingungsmöglichkeit, wird diese direkt in die Kommandozeile eingefügt, inklusive eines Leerzeichens um möglicherweise die Eingabe eines Parameters zu beschleunigen.

Existieren mehrere Möglichkeiten den Text zu komplettieren, so wird dem Benutzer eine Liste der Möglichkeiten präsentiert. Um zu vermeiden, dass der Benutzer mit einer Vielzahl an Möglichkeiten *erschlagen* wird, erfolgt diese Ausgabe nur, wenn die Menge der Möglichkeiten ein bestimmtes Limit nicht überschreitet. Ist dies der Fall, wird ersatzweise eine Nachricht ausgegeben, die den Benutzer informiert.

Der einfache Algorithmus wird in Abb. 6.8 skizziert.

```
def autocomplete(text):
    possibilities = []
    for command in commands:
        if command.startswith(befehl):
            possiblities.append(befehl)
    if not possibilities:
        inform_user('no completion')
    elif len(possibilies) is 1:
        complete(possibilities[0])
    elif len(possiblities < THRESHOLD):
        list_for_user(possibilities)
    else:
        inform_user('many completions possible')
```

Abbildung 6.8.: Vereinfachter Python-Code des Autovervollständigungsalgorithmus

6.3.2. Vervollständigung von Animationsnamen

Bislang können nur die Kommandos an sich, nicht aber ihre Parameter vervollstänidgt werden. Besonders störend ist dies beim Laden von bereits existierenden Animationen mit dem Befehl `load`, da die Animationsnamen mitunter recht lang ausfallen können und auch nicht so einfach zu merken sind, wie die begrenzte Menge an verfügbaren Kommandos.

Es ist aber relativ einfach den Algorithmus zur Vervollständigung von Befehlen auf Animationsnamen zu erweitern. Dazu muss eine Liste sämtlicher Animationsdateien im Suchpfad erstellt werden, welche geladen werden könnten. Diese wird dann als Menge der Vervollständigungsmöglichkeiten anstelle der bekannten Befehle im selben Algorithmus verwendet.

Es ist mit diesem Verfahren auch möglich, die Parameter anderer Befehle zu komplettieren. Allerdings wird im Rahmen dieser Arbeit aus Zeitgründen darauf verzichtet. Versucht der Benutzer den Parameter eines Befehls zu komplettieren, für den derzeit noch keine Komplettierung implementiert ist, dann erhält er eine Benachrichtigung darüber.

6.4. Kommandohistorie

Neben einer Autovervollständigung ist auch die Möglichkeit die Historie *durchzublättern* ein Feature, das von einer Kommandozeile erwartet werden kann.

Bereits die Ausgangsversion, das *Record-Script* bietet eine rudimentäre Historie, durch den einfachen Umstand, dass der Bildschirminhalt nie geleert wird. Es ist damit aber nicht möglich vergangene Eingaben wieder aufzurufen, um sie zu wiederholen oder anzupassen.

Mit *Urwid* haben wir außerdem eine Bibliothek eingeführt, welche den Bildschirminhalt imnmer wieder aktualisiert, sodass diese Grundfunktionalität zunächst verloren gegangen ist. Dieser Umstand war im Schritt 6.1.2 zunächst verschwiegen worden, um dort nicht zu tief in das Thema einzusteigen.

In diesem Entwicklungsschritt wollen wir also nicht nur die Historie wieder einsehbar machen, sondern das Konzept der Kommandozeile dahingehend aktualisieren, dass auch die Wiederverwendung von Eingaben möglich ist.

Sämtliche Ein- und Ausgaben der ***Record-UI*** werden einfach an eine Liste angefügt die stetig nach unten wächst. Diese Liste wird in einem Bereich angezeigt, den wir im folgenden *Ausgabehistorie* nennen. Zusätzlich wird aber jede Eingabe-Zeile an eine interne Histore angefügt. Diese nennen wir *Kommandohistorie*.

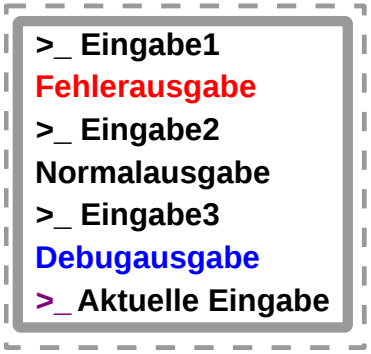


Abbildung 6.9.: Ansichtkonzept Konsolenhistorie incl. Eingabezeile

Mit den Tasten ↑ bzw. ↓ kann die Kommandohistorie rück- bzw. vorwärts *durchblättert* werden. Dabei wird der aktuelle Inhalt der Kommandozeile durch die jeweils nächste Eingabezeile aus der Historie ersetzt. Ist das Ende der Historie in der Vergangenheit erreicht, erhält der Nutzer eine Ausgabe, damit er davon Kenntnis erlangt. Wurde mehrfach hintereinander ein Befehl verwendet, so findet er sich auch mehrfach in der Historie. In anderer Richtung ist das Ende der Historie dadurch eindeutig gekennzeichnet, dass das Eingabefeld der Befehlszeile wieder geleert wird.

Was nun aber wenn der Benutzer einen Teil der Ausgabehistorie betrachten möchte, der weiter zurück liegt als die Ansicht erlaubt?

Dieser Anwendungsfall tritt eher selten auf, meist wenn Debug-Ausgaben direkt in die Konsole ausgegeben werden, um ein Problem zu lösen. Es ist dann notwendig die Historie *scrollen* zu können. *Urwid* unterstütz bereits das Scrollen der Historie mit den Pfeiltasten, wenn diese ausgewählt ist. Die Auswahl kann mit der Maus erfolgen, oder aber über Tastendruck in der Konsole. Als Taste zum Wechsel aus der Eingabezeile in die Historienliste wird `shift`+↑ vorgesehen, da ↑ ja bereits verwendet wird.

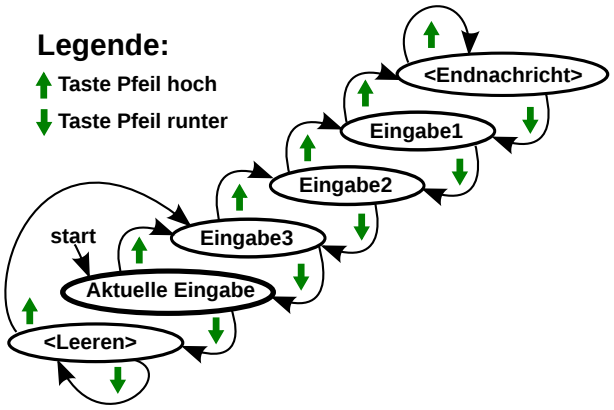


Abbildung 6.10.: Zustandsdiagramm der Historienfunktion zu Abb. 6.9

Die Möglichkeit, die Ausgabehistorie zu *Scrollen*, ist eher für fortgeschrittene Benutzer vorgesehen, es kann aber passieren, dass ein Benutzer diese Funktion versehendlich entdeckt, z.B. weil er als neuer Benutzer mit der Maus auf die Ausgabehistorie satt auf die Eingabezeile klickt.

Daher wird der *Cursor* wieder automatisch in die Eingabezeile verschoben, wenn ein von $\boxed{\uparrow}$ bzw. $\boxed{\downarrow}$ abweichender Tastendruck erfolgt. Handelt es sich um ein darstellbares Zeichen, wird dieses direkt an die aktuelle Eingabe angehängt. Dies ist im übrigen auch für die erfahrenen Benutzer bequem, die so schnell die Ausgabehistorie wieder verlassen können.

Das gesamte Verhalten der Kommandozeile ist damit weitestgehend konsistent mit den verbreiteten Kommandozeilen unter Linux.

6.5. Prüfung der Nutzereingaben

Die Daten, welche der Benutzer in die Formularfelder der Keyframeanzeige eingeben kann, werden derzeit nicht geprüft, dies soll geändert werden. Zwar ist das Framwork robust gegenüber fehlerhaften Werten, es können schließlich auch die Animationsdateien von Hand bearbeitet werden, doch ist es wünschenswert, wenn ein Fehler möglichst früh bemerkt wird, um die Arbeit zügig und frustrationsfrei zu gestalten.

Idealerweise soll es gar nicht erst möglich sein fehlerhafte Eingaben zu tätigen. Dazu wird zunächst an den Eingabefeldern selbst die Menge der überhaupt akzeptierten Zeichen auf das Minimum begrenzt. Dies hat auch einige technische Vorteile, auf die hier aber nicht weiter eingegangen wird.

Aber auch mit den erlaubten Schriftzeichzen ist es möglich, völlig unsinnige Eingaben zu

erzeugen, Beispielsweise `-1-` oder `12.0.5`, schließlich müssen auch negative und nicht-ganzzahlige Motorwinkel eingegeben werden können.

Es wird daher bei Eingabe eines erlaubten Zeichens geprüft, ob die neue Ergänzung das Eingabefeld ungültig machen würde. Falls ja wird die Eingabe verworfen. Tippfehler sind sehr häufig, daher wird der Benutzer hierüber nicht gesondert unterrichtet.

Zusätzlich soll sichergestellt werden, dass die eingegebenen Zielwinkel für die Motoren tatsächlich im vom Roboer anfahrbaren Intervall liegen. Das erlaubte Intervall wird der Konfigurationsdatei des Roboters entnommen. Unter- bzw. überschreitet eine Eingabe das erlaubte Intervall, so wird der Wert im Eingabefeld automatisch auf den nächstliegenden erlaubten Wert korrigiert.

Der Benutzer erhält dadurch direkt eine Berichtigung, die er sehen und verstehen kann. Er wird aber nicht weiter im Arbeitsfluss behindert sondern lernt, dass er sich auf die Software verlassen kann und nicht fürchten muss schädliche Eingaben zu tätigen. Das Verhalten lässt sich auch intuitiv nutzen, um einen Motorwinkel auf den Maximalwert zu setzen ohne diesen zu kennen, indem man z.B. `999` als Zielwert eingibt.

Mit diesen Änderungen ist es *fast* unmöglich ungültige Eingaben zu verursachen, mit zwei Ausnahmen. Der Nutzer muss das Eingabefeld restlos leeren können, um eine neue, gültige, Eingabe zu tätigen und er muss in der Lage sein ein einzelnes Minuszeichen [-] zu schreiben, um eine negative Zahl zu beginnen. Beide Zustände sind keine gültigen Werte.

Es wäre nun möglich dem Benutzer die Navigation aus dem Feld zu verbieten, solange die Daten ungültig sind. Diese Art der Modalität wäre für einen unerfahrenen Nutzer jedoch schwer zu verstehen und könnte Frustration aufbauen.

Auch wäre es möglich das Feld bei Navigation wieder auf den Ursprungswert zurück zu setzen, aber der Benutzer hat die Eingabe vermutlich in der Absicht getätigt den Inhalt zu verändern und ein Zurücksetzen des Wertes könnte Fehler provozieren, die der Nutzer nur schwer nachvollziehen kann.

Stattdessen soll dem Benutzer erlaubt werden das Feld trotz ungültigem Zustands zu verlassen, damit der Fehler aber bemerkt werden kann wird das Feld, und gleichermaßen auch der zugehörige Keyframe farblich als Fehlerstelle gekennzeichnet, damit der Nutzer selbst den Fehler beheben kann. Im gleichen Zuge wird das Speichern der nun ungültigen Animation unterbunden und nur eine Fehlermeldung ausgegeben.

Als weiteren Nutzen der Fehleranzeige, können wir nun auch neu geladene Animationen auf ihre Korrektheit prüfen, und sämtliche Fehlerstellen kennzeichnen, damit Animatio-

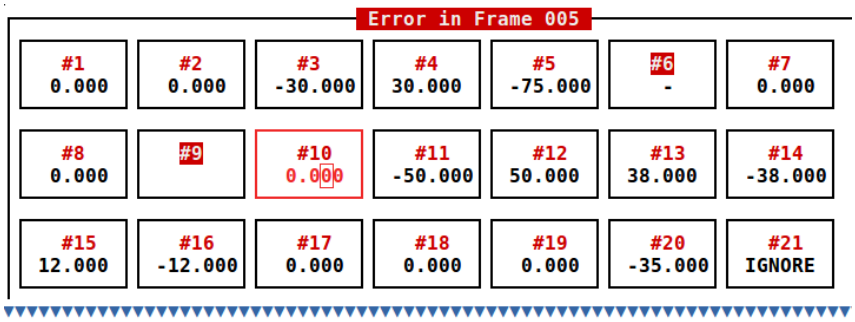


Abbildung 6.11.: Ein Keyframe mit zwei fehlerhaften Datenfeldern

nen, die bereits vor dem Laden in fehlerhafter Form vorliegen, einfach korrigiert werden können.

6.6. Automatische Speicherung

Bereits die Ausgangsversion, das *Record-Script*, bietet die automatische Speicherung der Animation bei Verwendung des Befehls `record` als Datei des Namens `Backup.json`.

Dieses Verhalten wurde in die ***Record-UI*** übernommen, beschränkt sich aber ausschließlich auf das `record` Kommando. Es soll daher erweitert werden, um die Fehlertoleranz der Software zu erhöhen.

Grundsätzlich soll eine automatische Speicherung jedes mal erfolgen, wenn die zugrundeliegende Animation verändert wird. Also auch bei allen entsprechenden Funktionen, die im Folgenden noch beschrieben werden. Eine automatische Speicherung findet nicht statt, wenn der Zustand der Animation ungültig ist, was wie in Abschnitt 6.5 beschrieben, in Einzelfällen passieren kann. So soll sichergestellt werden, dass die `backup.json` immer eine valide Animation ist.

6.7. Rückgängig machen von Nutzeraktionen

Die Möglichkeit, die eigenen, fehlerhaften Aktionen zu revidieren, ist essentiell für eine nutzerfreundliche Anwendung. Abgesehen von der Arbeitserleichterung nimmt dies dem Benutzer auch ein Stück weit die Angst einen Fehler zu begehen. Die Funktion ist auch ein Weg, die in der [7] geforderten Fehlertoleranz zu realisieren.

Wir möchten also den Befehl `undo` realisieren, der die zuletzt getätigte Aktion rückgängig macht. Es ergibt sich jedoch die Frage, ob wirklich alle Aktionen rückgängig gemacht werden sollten. Ist beispielsweise der Wechsel der Ansicht eine Aktion des Benutzers die durch einen Undo-Befehl reversibel sein sollte?

Wirklich kritische Änderungen sind solche, die den Zustand der zugrunde liegenden Animation verändern. Also genau die, die eine automatische Speicherung auslösen. Für genau diese Aktionen soll auch die Undo-Funktion greifen.

Dabei erhält der Nutzer bei Eingabe des Befehls `undo` zweierlei Rückmeldungen:

1. Welche Aktion wurde rückgängig gemacht?
2. Was ist die neue *zuletzt* getätigte Aktion ?

Wir erhalten dabei ein neues Problem: Was wenn der Benutzer irrtümlich eine Aktion zurück genommen hat?

Dafür führen wir als weiteren Befehl `redo` ein, welcher eine zuvor zurück genommene Aktion wiederholt, wobei auch hier die Rückmeldung in der Konsole erfolgt, welche Aktion *wiederholt* wurde.

Zu guter letzt erlauben wir uns noch eine Vereinfachung für Benutzer, die mehrere Schritte Rückgängig machen möchten und geben den Befehlen `undo` und `redo` einen optionalen Parameter, mit dem die Anzahl der zurückzunehmenden bzw. zu wiederholenden Schritte angegeben werden kann.

6.8. Anpassbares Farbschema

Die meisten modernen Terminalanwendungen unterstützen die Verwendung von Farben, wobei die Menge der zur Verfügung stehenden Farben variiert.

Im Allgemeinen lassen sich zwei Gruppen von Terminalnutzern beobachten, solche, die ein helles Farbschema für ihre Terminalanwendung nutzen (meist schwarze Schrift auf weißem Grund) und solche die einen dunklen Hintergrund und eine helle Vordergrundfarbe bevorzugen. Den Nutzern von Terminals stehen aber meist weit mehr Konfigurationsmöglichkeiten zur Verfügung. Beispielsweise die Anpassung von Schriftart und Schriftgröße, aber auch Transparenz des Hintergrundes. Auf die Terminaleigenschaften an sich hat die ***Record-UI*** technisch keine Einflussmöglichkeit, lediglich die Farben können dem Terminal *vorgeschlagen* werden.

Werden die von der ***Record-UI*** verwendeten Farben dem Benutzer aufgezwungen, kann dies das Nutzungserlebnis in erheblichem Maße negativ beinflussen, wenn sich die Anwendung nicht in das Bild einfügt, welches das System des Anwenders üblicherweise bildet. Daher greift die ***Record-UI*** im Standardfarbschema bereits ausgiebig auf den *Default-wert* für die Textfarbe und den Hintergrund zurück, so dass die Standardeinstellung im

Abbildung 6.12.: Gegenüberstellung zweier verschiedener Konsolenkonfigurationen, bei gleichbleibendem Farbschema der ***Record-UI***

Terminal des jweiligen Nutzers verwendet wird. Davon abweichende Farben müssen jedoch explizit spezifiziert werden, was nicht immer eine harmonische Kombination mit dem Farbthema des Terminals bildet, wie in Abbildung 6.12 zu sehen ist.

Auch sei darauf hingewiesen, dass einige Nutzer aufgrund von Einschränkungen des eigenen Sehvermögens gezwungen sein könnten, die Farbgebung anzupassen, nicht zuletzt deshalb ist die Individualisierbarkeit von Software auch eine Forderung aus der DIN EN ISO 9241-110:2006 [7].

Das Farbschema der ***Record-UI*** soll daher über ein Konfigurationsdatei im yaml-Format angepasst werden können. Ein entsprechender Eintrag dort sieht beispielsweise folgendermaßen aus:

```
- ['alert', 'white', 'dark red', 'blink, bold, standout']
```

Dieser Eintrag weist dem Farbbegriff *alert* die Farbe weiß auf rotem Hintergrund zu, und setzt einige Optionen, die von den Terminalumgebungen berücksichtigt werden *können*.[1] Der Syntax ist derart gewählt, dass die Einträge nach dem Einlesen direkt *Urwid* als *Palette* übergeben werden können (siehe dazu [18]).

Manche Terminals unterstützen 256 Farben, für diese kann man optional den Eintrag erweitern , z.B.:

```
- ['alert', 'white', 'dark red', 'blink, bold, standout', '#EEE', '#0E0']
```

Die letzten beiden Farbangaben werden als Vorder- bzw. Hintergrundfarbe verwendet, wenn das Terminal 256 Farben unterstützt. Andernfalls sorgt die Implementation von *Urwid* dafür, dass sinnvolle Fallbacks verwendet werden. So ist grundsätzlich auch die Verwendung in monochromen Terminals möglich. In diesem Falle kann je nach Fähigkeit des Terminals beispielsweise der Parameter *standout* genutzt werden, um den Text

[1] Es gibt keine Möglichkeit eine bestimmte Darstellung zu erzwingen, gewissermaßen handelt es sich nur um einen Vorschlag an die Terminalumgebung.

in Invertierter Farbe anzuzeigen, damit selbst in monochromen Terminals noch die Aufmerksamkeit des Nutzers auf eine bestimmte Stelle der UI gelenkt werden kann (vgl. 6.11).

Erweiterungsmöglichkeiten Eine Anpassung des Farbschemas aus der ***Record-UI*** selbst heraus ist derzeit nicht vorgesehen, kann aber in späteren Versionen implementiert werden, z.B. mit einem Befehl zum Laden eines Farbschemas der Form `theme <name>`.

Sinnvoll könnte auch ein Kommando der Form `dark` bzw. `light` sein, mit dem der Benutzer unmittelbar auf ein *dunkles* bzw. *helles* Standard-Farbthema umsteigen kann, um den beiden eingangs genannten Gruppen von Nutzern noch besser gerecht zu werden. Darüber hinaus wäre ein schneller Wechsel vorteilhaft, um auf sich änderndende Lichtverhältnisse (Sonne auf dem Bildschirm, Dunkelheit) reagieren zu können.

6.9. Neue Kommando-Architektur

Wir haben bis hier her im Wesentlichen ein UI-Konzept entworfen, welches geeignet ist, die größten Schwächen des alten *Record-Scripts* durch eine pseudo-graphische Oberfläche zu beheben.

Im Folgenden geht es darum, welche Funktionen man unter Verwendung der Kommandozeile bieten kann, um die Nutzung der ***Record-UI*** zu verbessern. Bevor wir aber beginnen neue Kommandos einzuführen, wollen wir uns einmal anschauen wie bisher die Aufteilung der Softwarekomponenten aussieht.

Wir haben, im Wesentlichen, zwei Komponenten. Die neue ***Record-UI*** , in der Datei `ui.py` und das alte *Record-Script*, dessen Funktionen weitestgehend in der `record.py` erhalten bleiben. Die UI erlaubt über die Keyframeanzeige eine Manipulation des Zustands vom *record*. Alle Befehle, die in die Kommandozeile der UI eingegeben werden, werden an das *record* zur Ausführung übergeben.

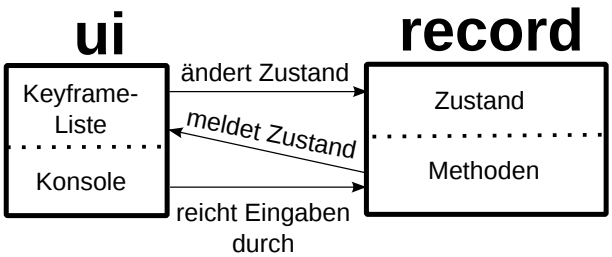


Abbildung 6.13.: Bisheriger Aufbau der ***Record-UI*** (vereinfacht)

Dieser Aufbau ist einfach, aber langfristig nicht sinnvoll. Wir möchten eine Trennung zwischen Nutzeroberfläche und *Geschäftslogik* erreichen. Historisch gesehen gab es beim *Record-Script* nie einen Unterschied zwischen *frontend* und *backend*, künftig soll der Programmteil *record* nur noch den Zustand der Animation, sowie allgemein gehaltene Methoden zur Veränderung des Animationszustandes anbieten.

Die Eingaben in die Kommandozeile der UI sollen daher zunächst von dieser verarbeitet werden, bevor davon ausgehend entschieden wird, welche Methoden des *record* mit welchen Parametern aufgerufen werden.

Um auch innerhalb der UI eine übersichtliche und wiederverwendbare Codeaufteilung zu erhalten, wird dafür eine neue Komponente geschaffen, der sog. *Commander* in der Datei `commands.py`.

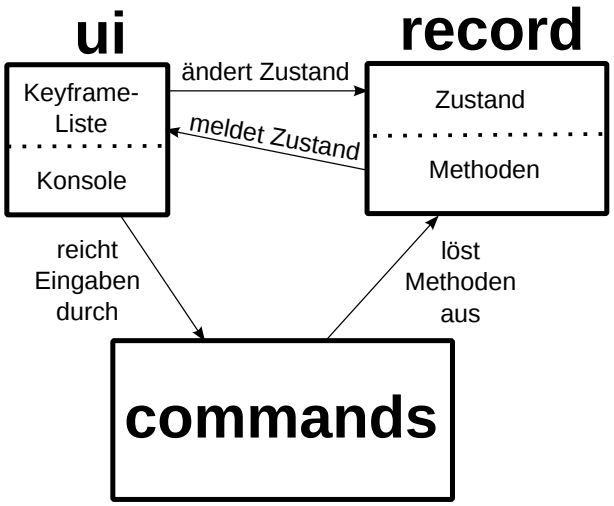


Abbildung 6.14.: Neuer Aufbau der ***Record-UI*** (vereinfacht)

Diese neue Komponente kümmert sich darum Nutzereingaben zu prüfen, zu verarbeiten und die Entscheidung zu treffen, welche Methoden am record aufgerufen werden. Dazu erhält sie von der Kommandozeile der UI die Nutzereingabe, die UI selbst verarbeitet die Eingabe nicht.

So erhalten wir drei Komponenten die sich mit vertretbarem Aufwand von einander trennen lassen und für die Entwicklung einer neuen Anwendung wiederverwendet werden können. Beispielsweise kann der neue *Commander* verwendet werden, um eine Kommandozeile in einer ganz neuen graphischen Oberfläche zu realisieren, die vom hier vorgestellten terminalbasierten Ansatz abweicht.

6.10. Hilfefunktion

Mit der alten Kommandozeile ist die bisher einzige Form der Nutzerhilfe eine Ausgabe im Fehlerfall, die möglicherweise notwendige Informationen über einen Befehl preisgibt.

Insbesondere bei CLI-Anwendungen ist eine komplexe Hilfefunktion notwendig, da optischen Anhaltspunkte fehlen, ein Problem intuitiv zu lösen. Zwar kann man bis zu einem gewissen Grad intuitiv erahnen wie ein Befehl wahrscheinlich lauten müsste, dann mangelt es jedoch weiter an der Kenntnis über die genauen Funktionsweise und über die erwarteten Parameter. Hier soll der Befehl `help` abhilfe schaffen und dem Nutzer eine Möglichkeit geben schnell nachschlagen zu können, wie ein Kommando aufgebaut ist, oder Informationen zu bestimmten Themengebieten zu erhalten.

Die meisten gebräuchlichen Kommandozeilen bieten auch tatsächlich eine Hilfefunktion, so besitzen unter Linux die meisten Programme eine *Manpage* die mit dem Befehl `man <topic>` aufgerufen werden kann. Im Kommandozeileneditor *vim* heißt dieser Befehl `help <topic>`, im Kommandozeileninterpreter von Python verwendet man `help(<topic>)`. Da bereits die Tastenkürzel an *vim* angelehnt wurden wird auch der Hilfe-Befehl entsprechend benannt, zumal `help` eine Art Quasi-Standard in diesem Bereich zu sein scheint.

Die Hilfebeiträge sollen dabei in einer externen Datei gespeichert sein, wo sie einfach erweitert und angepasst werden können. Dafür wählen wir das *YAML*-Format.

Die Ansicht des formatierten Hilfetextes soll an Stelle der Keyframeliste erfolgen, damit die Kommandozeile weiter verfügbar ist und die Steuerbarkeit gewährleistet bleibt.

Eine besondere Formatierung soll für Kommandos erfolgen. Dort interessiert es zumeist, wie der Aufbau der Parameter zu gestalten ist. Daher wird die folgende Syntax gewählt, die an die Beschreibungen der Befehle der *Unix-Shell* angelehnt sind.

```
kommando <zwingender-parameter> [optionaler-parameter]
```

```
author:
  usage: "author [authorname(s)]"
  help: |
    Setzt die Autoreninformationen auf den/die angegebenen Namen
    oder zeigt die aktuellen Autorinformationen wenn
    keine Parameter angegeben werden.
    .. hint:: Die Namen werden *nicht* in Anführungszeichen gesetzt.
```

Abbildung 6.15.: Ein Hilfe-Eintrag in der Quelldatei

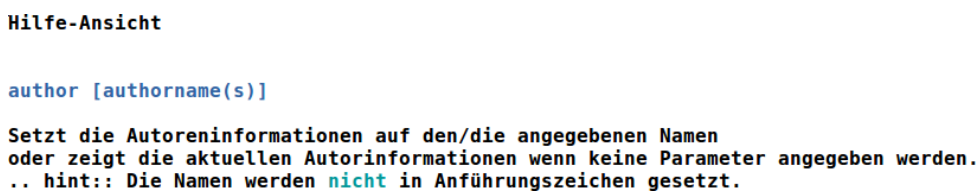


Abbildung 6.16.: Hilfe-Eintrag aus Abb.6.15 in der ***Record-UI***

In Abb. 6.15 ist das Format der Hilfedatei abgebildet, in Abb. 6.16 kann die fertige Übersicht in der ***Record-UI*** betrachtet werden.

Test auf Vollständigkeit Da die Dokumentation der Kommandos besonders wichtig ist, wird auch ein Unittest eingeführt, welcher prüft ob alle in der Datei `commands.py` spezifizierten Befehle jeweils einen Eintrag in der Hilfedatei haben.

6.10.1. Schwächen der Hilfefunktion

Eine mündliche Befragung von Nutzern der Record-UI hat ergeben, dass diese offenbar die verfügbare Online-Dokumentation gegenüber der integrierten Dokumentation bevorzugen.

Die Online-Dokumentation bietet unbestreitbare Vorteile gegenüber der integrierten Hilfe. Zum einen ist die Darstellung wesentlich ansprechender, als sie in einem Terminal geleistet werden kann. Zum anderen unterstützt die Online-Dokumentation die Verlinkung, mit anderen Themen und Websites aus dem WWW sowie eine Volltextsuche. Gerade die Volltextsuche ist ein Merkmal, welches die interne Hilfe gut ergänzen würde.

6.10.2. Doppelte Dokumentation

Aktuell existieren zwei verschiedene Möglichkeiten der Hilfe. Die mit *Sphinx* generierte Online-Dokumentation und die hier beschriebene Hilfefunktion. Beide Dokumentationen sind disjunkt, obwohl der Inhalt teilweise redundand ist.

Mittelfristig wäre daher erstrebenswert, dass beiden Dokumentationen die selbe Quelle zugrunde liegt. [2]

6.11. Keyframes spiegeln

Die bei den *Hamburg Bit-Bots* verwendeten Roboter sind humanoid und als solche sind sie symmetrisch aufgebaut. Häufig ist es gewünscht, dass Teile beider Körperhälften einander entsprechen, z.B. wenn es um einen sicheren Stand auf glattem Untergrund geht.

[2] Tatsächlich existieren dazu auch bereits Programmiervorleistungen

Verschiedene Faktoren erschweren es aber, einen vollständig symmetrischen Stand mit dem Roboter zu erreichen, der dann aufgenommen werden kann.

Diese Faktoren sind:

- Unebener Untergrund
- Ungenauigkeit beim in Pose setzen mit der Hand
- Ungenauigkeit beim Erfassen der Motorpositionen

Die sich daraus ergebenden Abweichungen können beim Abspielen der Animation die Stabilität des Roboters gefährden. Besonders kritisch für die Stabilität ist meist das Abweichen von der Symmetrie. Liegt beispielsweise ein Fuß ungewollt etwas niedriger als der andere, neigt sich der Roboter und die Stabilität ist verringert. Wird nun die Position des einen Fußes auf die des anderen projeziert, steht der Roboter gerade. Die Stabilität wird daher oft sogar dann verbessert, wenn die leicht vom Idealwert abweichende Position eines Körperteils auf eine ideale Position projeziert wird.

Die Funktion einen Keyframe zu spiegeln ist somit eine ungemein nützliche. Auch Hohberg [9, S.17] nimmt sie in seine Anforderungen auf. Dem Benutzer soll daher das folgende Kommando bereitstehen:

`mirror <Zielframe> <Zieltag>`

Es soll eine Menge von Motoren die in `<Zieltag>` spezifiziert sind in dem Keyframe `Zielframe` auf ihren jeweiligen Gegenpart projeziert werden, sofern vorhanden.

Die erste Schwierigkeit die sich ergibt: Welcher Motor liegt einem gegebenen Motor gegenüber? Hier hilft die Konfigurationsdatei *joints.yaml* weiter. Speziell für diesen Verwendungszweck wird der Schlüssel `opposing` eingeführt, mit dem jedem Gelenk die id eines Gegenparts zugewiesen werden kann. Wird der Wert `0` zugewiesen, ist damit gekennzeichnet dass der Motor keinen Gegenpart besitzt und beim Spiegeln ignoriert wird.

Das zweite Problem ergibt sich unmittelbar aus der Chiralität des Roboters selbst. Die meisten Motoren des Roboters sind chiral, also spiegelbildlich, eingebaut. Dies hat zur Folge, dass eine positive Bewegung des einen Motors einer negativen Bewegung seines Gegenparts entspricht. Es ist aber leider nicht möglich einfach *alle* Werte zum spiegeln mit -1 zu multiplizieren, da es einige Motoren gibt, die von dieser Regel abweichen, beim *Darwin-OP* beispielsweise die `HipRoll`-Motoren.

Darum wird ein zweiter Schlüsselwert für die Gelenke notwendig: `inverted`, welcher die Boolschen-Werte `True` und `False` annehmen kann, um anzugeben, ob die Drehrichtung des Motors gegenüber seinem Gegenpart invertiert ist.

Mit diesen zusätzlichen Daten, die über den Roboter bereit stehen, ist es nun trivial die Spiegelfunktionalität zu implementieren. Als zusätzliches Entgegenkommen für den Nutzer wird es auch möglch statt einer Keyframenummer für `<Zielframe>` auch das Spezialziel `all` anzugeben, um die Spiegelung auf sämtliche Keyframes der Animation anzuwenden.

6.12. Metainformationen für Animationen

Die Animationserstellung für Roboter ist ein Prozess. Eine Animation wird oft nicht einfach an einem Tag erstellt sondern über einen längeren Zeitraum entwickelt. Bei Veränderungen am Roboter muss oftmals eine Animation angepasst werden, damit der Roboter die Animation ausführen kann. Diese Anpassung kann teilweise mit großem zeitlichen Abstand zur Erstellung der Originalanimation erforderlich werden.

Da meist mehrere Personen in einem Robotikprojekt involviert sind, ist es also notwendig die Arbeit an einer Animation zu dokumentieren. Dazu sollen nun einige Metadaten zusammen mit der Animation gespeichert werden.

Name Jede Animation hat in der Ausgangsversion nach dem Speichern bereits einen Namen, der dem Dateinamen der Speicherdatei entspricht. Ein solcher Name ist nicht notwendigerweise eindeutig, es können mehrere Dateien des gleichen Namens an unterschiedlichen Orten liegen.

Autoreninformation Der Benutzer *kann* das Feld `Author` mit einem oder mehreren Namen von Personen befüllen, die an der Erstellung der Animation beteiligt waren. So können ihnen später gezielt Rückfragen gestellt werden.

Versionsinformation Jede gespeicherte Animation erhält automatisch eine Versionsnummer, die beim Speichern einer neuen Version inkrementiert wird. Während dieses einfache Verfahren alleine ganz offensichtlich nicht geeignet ist, sich verzweigenden Entwicklungen gerecht zu werden, ist es bereits eine große Hilfe, um bei linearen Entwicklungsverläufen zu entscheiden, welche der vorliegenden Versionen die *aktuellere* ist.

Bearbeitungsdatum Auch das Datum der letzten Bearbeitung wird automatisch in die .json-Datei der Animation eingefügt. Es bleibt so erhalten auch wenn das Bearbeitungsdatum der Speicherdatei z.B. durch einen Kopiervorgang verloren geht.

Hostname Zusätzlich wird automatisch der Hostname des Rechners gespeichert, auf dem die ***Record-UI*** zur Ausführung gebracht wird. Bei den Robotern der *Hamburg Bit-Bots* entspricht der Hostname dem (eindeutigen) Namen des Roboters, sodass eine Zuordnung zur Hardware hergestellt werden kann, auf der die Animation aufgenommen wurde. Dies ist wichtig, um im Zweifelsfall Fehler zu identifizieren, welche durch (teils minimale) Unterschiede in der verwendeten Hardware entstanden sind.

6.12.1. Umsetzung in der UI

Es werden einige neue Befehle eingeführt mit denen der Nutzer die Metadaten über die Kommandozeile auslesen, bzw. setzen kann.

```
author [autor(en)]
desc [beschreibung]
meta
```

Die ersten beiden Befehle geben bei Aufruf ohne Parameter die jeweilige Information in der Konsole aus. Erfolgt der Aufruf mit zusätzlichen Parametern, werden diese als Text interpretiert, mit dem das Datenfeld belegt wird.

Der dritte Befehl ist speziell, er öffnet eine neue Ansicht an Stelle der Keyframeliste, in der sämtliche Metadaten ausgelesen, sowie die nicht automatisch generierten Felder `author` und `description` bearbeitet werden können.

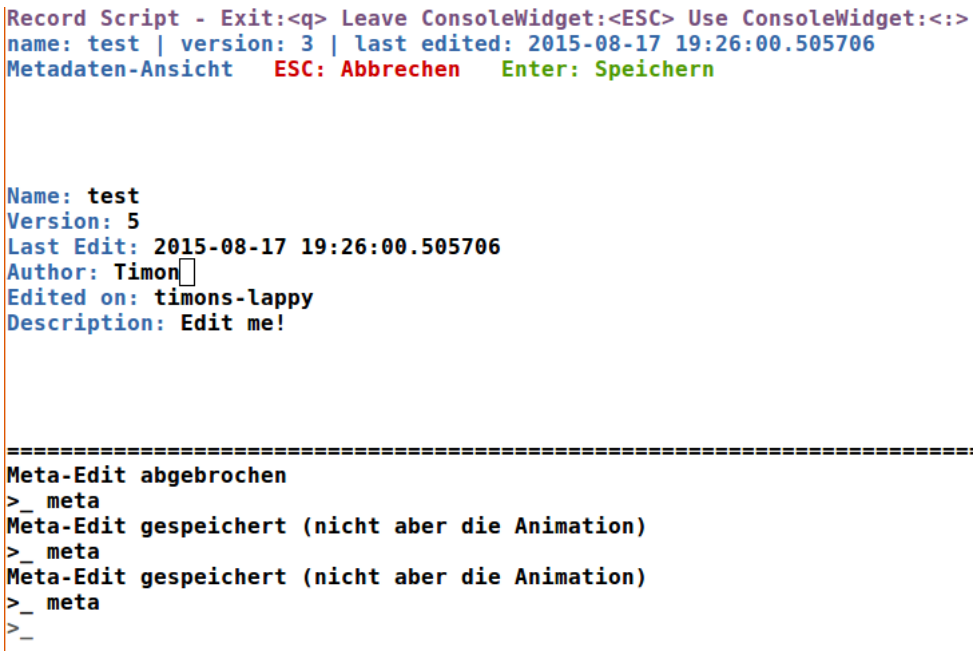


Abbildung 6.17.: ***Record-UI*** mit geöffneter Metadaten-Ansicht

6.13. Zusatzinformationen zu den Motoren

6.13.1. Anzeigen der Motorposition im Roboter

Während ich dem Argument von Hohberg [9] wiedersprechen möchte, dass eine Visualisierung des Roboters es praktikabel macht, die Animationserstellung ohne einen Roboter zu vollziehen, ist unzweifehlaft zu bemerken, dass die ***Record-UI*** es dem Nutzer schwer macht nachzuvollziehen, wo ein gegebener Motor im Roboter tatsächlich verortet ist.

Es stellt sich daher die Frage, ob es eine Möglichkeit gibt, trotz der limitierenden Terminalumgebung, die Position der Motoren im Roboter zu verdeutlichen. Eine informative Grafik scheidet auf den ersten Blick aus, dies liegt aber daran, dass wir heutzutage graphische Oberflächen so sehr gewohnt sind, dass wir uns nicht vorstellen können ohne sie zu arbeiten.

Auszug aus der

Schreibmaschinen-Zeitung

Monatsschrift für das gesammte Schreibmaschinenwesen.

Herausgeber: **Otto Burghagen**, Handelsschuldirector. — Redaction und Expedition: HAMBURG, Bergstr. 21
(Post-Zeitungs-Preisliste 11. Nachtrag No. 6563a.)

Der **Abonnementspreis** beträgt halbjährlich 1 Mark ohne Bringerlohn. Bestellungen nehmen alle Postanstalten, Buchhandlungen, sowie die Expedition entgegen. Zusendung unter Streifband 1,30 Mark im Inlande, und 1,50 Mark im Auslande (Weltpostverein).

Insertionsgebühr 30 Pfennige die Petitzeile (Stellengesuche 10 Pfennige pro Zeile). — 1/2 Seite 60 Mark. — 1/1 Seite 35 Mark. — 1/2 Seite 20 Mark. — Wiederholungen nach Uebereinkunft. Annoncen an bevorzugter Stelle unterliegen einer Zuschlagsgebühr. Vorausbezahlung an den Herausgeber. Erfüllungsort Hamburg.

No. 3. **Hamburg, 15. September 1898.** **I. Jahrg.**

Fürst Bismarck.

Gezeichnet mit der **„Jewett"-Schreibmaschine** (Schreibmaschinen-Fabrik Sundern, Westfalen).
Das Bild ist ohne jede Nachhülfe auf der „Jewett"-Schreibmaschine hergestellt worden. Unsere Holzschnittzeichnung ist eine auf photographischem Wege hergestellte getreue Nachbildung der ursprünglichen Zeichnung, welche uns im Originale vorgelegen hat. Man kann deutlich die einzelnen Satzzeichen erkennen, (Punkte, Striche, Kommas etc.), aus denen die Zeichnung sich zusammensetzt. Jedenfalls illustriert diese Zeichnung die ungemein leichte Handhabung der „Jewett"-Maschine.

Abbildung 6.18.: ASCII-Art von 1898.
Bildquelle: PA AA, RZ 607, R 138985
Politisches Archiv des Auswärtigen Amts

Seit es technische Hilfen zum Textsatz gibt, haben Menschen diese Methoden auch zweckentfremdet, um pseudo-graphische Inhalte zu produzieren (siehe Abb.6.18).

Das Stichwort lautet *ASCII-Art* und um die Möglichkeit der graphischen Darstellung im Textterminal zu demonstrieren, soll ein Befehl für die ***Record-UI*** eingeführt werden, der zumindest für einen Standard *Darwin-OP* Roboter die Position der Motoren hinreichend demonstriert.

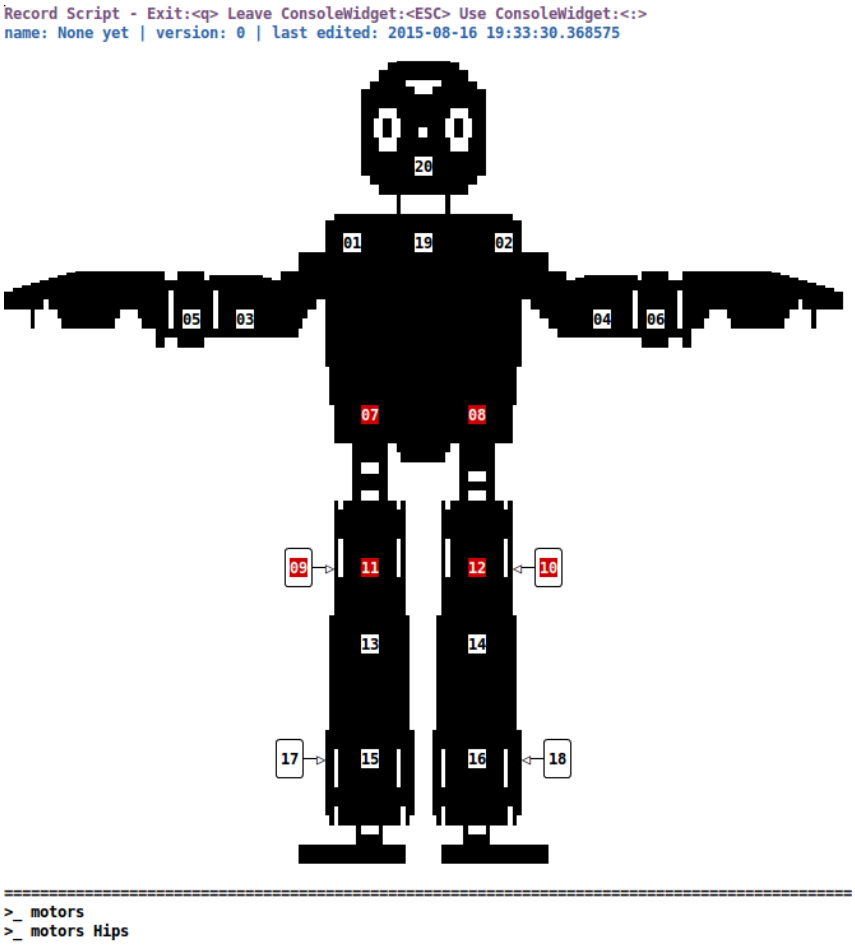


Abbildung 6.19.: Output des Befehls `motors Hips`
vgl. Abb. 3.2

Bei Eigabe des Befehls `motors [Tag]` wird eine ASCII-Art-Repräsentation eines Darwin-OP Roboters gezeigt. Bei gegebenem `Tag` werden die ID's der zu dem Tag gehörenden Motoren farblich hervorgehoben (siehe Abb. 6.19).

6.13.2. Motorinformationsübersicht

Wie im vorigen Abschnitt geschildert existiert mit der `joints.yaml` eine Datei, in der verschiedene Metainformationen über die Motoren bereit gestellt werden.

Es ist daher nicht schwierig diese Informationen auch für die Nutzer der ***Record-UI*** aufzubereiten, damit diese sich direkt in der Animationserstellung Daten zu den Motoren anzeigen lassen können.

```
Record Script - Exit:<q> Leave ConsoleWidget:<E
name: None yet | version: 0 | last edited: 2015
Name: LKnee
ID: 14
Tags:
    legs
    lleg
    knees
    left
    14
opposing motor: 13
Motor IS inverted against the opposing motor


================================================
>_ motorinfo LKnee
```

Abbildung 6.20.: Output des Befehls `motorinfo LKnee`

Das Prinzip funktioniert dabei ähnlich der Hilfefunktion. Der Nutzer verwendet den Befehl `motorinfo <Tag>` und es wird mit Hilfe der Metadaten aus der `joints.yaml` eine Übersicht erstellt, die die Daten aller diesem Tag zugeordneten Motoren in einem Übersichtsfenste an Stelle der Keyframeliste anzeigt.

6.13.3. Kurze Taginformation

Während der zuvor beschriebene Befehl `motorinfo` sehr detailierte Informationen über die Motoren eines Tags liefern kann, stellt sich der Benutzer oft einfach nur die Frage, welche Motornummern er in einem Keyframe editieren muss, wenn er beispielsweise den linken Fuß verändern möchte.

Hierbei soll ein einfacherer Befehl Hilfestellung leisten, der die mit einem Tag assoziierten Motoren benennt. Dafür schaffen wir das Kommando `id <Tag>`.

```
>_ id Left
[2, 4, 6, 8, 10, 12, 14, 16, 18, 22, 24, 26]
>_ id LKnee
[14]
```

Abbildung 6.21.: Zwei verschiedene Ausgaben des Befehls `id`

Erweiterungsmöglichkeiten

Der Befehl `motors` kann derzeit nur einen Robotertyp abbilden und dient damit eher als Demonstrator. Aus Zeitgründen wird darauf verzichtet den Befehl für die anderen bei den *Hamburg Bit-Bots* gebräuchlichen Robotermodelle bereitzustellen.

6.14. Anfahren einer Pose

Ein weiteres Problem, das für den Benutzer gelöst werden soll, ist, dass er manchmal von einem bestimmten Keyframe in der Animation ausgehend arbeiten möchte, z.B. um Verbesserungen an dem Keyframe auszuprobieren.

Dafür wird der Befehl `pose <keyframenummer>` eingeführt, mit dem der Benutzer den Roboter eine bestimmte Pose, also das Standbild eines Keyframes, einnehmen lassen kann. Die Ablaufgeschwindigkeit des Keyframes wird dabei ignoriert, da es nicht darauf ankommt die Animation originalgetreu abzufahren und das Verletzungsrisiko durch eine niedrige Bewegungsgeschwindigkeit reduziert werden kann (siehe zum Verletzungsrisiko auch Abschnitt 8.5.1).

6.15. Abspielen einer Teilanimation

Für die Entwicklung einer Animation kann es sinnvoll sein nur einen Teil der Animation abzuspielen. Oft ist man sich im Verlauf der Animationserstellung noch unsicher, welche Teile der Animation behalten werden sollen und welche *wegoptimiert* werden können. Auch verbraucht das Abspielen einer aufwändigen Animation mitunter viel Zeit und es kann Sinn ergeben beim abspielen einen Teil auszulassen, von dem man bereits weiß dass er wie erwartet funktioniert.

Daher soll der Befehl `play` erweitert werden und als zusätzlichen Parameter eine Spezifikation annehmen welche Keyframes in die abzuspielende Animation mit aufgenommen werden sollen.

Dabei wird angenommen dass die abzuspielenden Keyframes in aufsteigender Reihenfolge als Kommaseparierte Aufzählung, als mit einem Minuszeichen getrennte *Range*, oder eine Mischung von beidem Spezifiziert werden (siehe Abb. 6.22).

```
play 1,3-5,7,9
```

Abbildung 6.22.: `play`-Befehl exklusiv für die Keyframes 1,3,4,5,7,9

6.16. Unterspezifizierte Motoren

Eine Anforderung, die bislang unberücksichtigt blieb, ist die Möglichkeit einen Motor in einer Animation unterspezifiziert zu lassen.

Beispielsweise soll es möglich sein eine Animation nur für den Bereich ab der Hüfte aufwärts zu erstellen, und so die Beine für die Verwendung durch einen Walking-Algorithmus *frei* zu halten. Das alte *Record-Script* bot eine solche Möglichkeit nicht, es wurden immer alle Motoren aufgenommen, wenn einzelene Motoren nicht spezifiziert werden sollten, dann mussten sie von Hand aus der entsprechenden Animationsdatei herausgelöscht werden.

Auch die ***Record-UI*** ermöglicht es nicht zu spezifizieren welche Motoren beim Kommando `record` erfasst werden. Zumindest soll aber die Möglichkeit geschaffen werden beim Bearbeiten der Keyframes einen Wert zu entfernen.

Daher wird die Möglichkeit geschaffen statt eines gültigen Motorwinkels ein `i` in das Datenfeld einzutragen, welches sodann durch den Wert `IGNORE` ersetzt wird. Letzteres soll der Selbstbeschreibungsfähigkeit dienlich sein.

6.17. Abfangen von Systemsignalen

Ein besonderes Problem, das bislang auftreten kann, ist, dass der Prozess der ***Record-UI*** von außen beendet wird.

Beispielhaftes Szenario: `Alice` arbeitet mit der ***Record-UI*** auf dem Roboter, `Bob` überspielt eine neue Software auf den Roboter ohne sich bewusst zu sein dass `Alice` noch mit der ***Record-UI*** arbeitet. Beim Überspielen der neuen Software wird automatisch ein `killall python` ausgeführt, um alle Prozesse auf dem Roboter neu zu starten[3] und `Alice` wird plötzlich und unerwartet aus der Arbeit an der ***Record-UI*** heraus gerissen und verliert möglicherweise ihren Arbeitsstand.

Daher soll die ***Record-UI*** nunmehr das Systemsignal `SIGTERM` abfangen und dem Benutzer die Möglichkeit geben zu entscheiden was geschehen soll. Dieses Verhalten ist von vielen Programmen, beispielsweise aus dem Office-Bereich bekannt, wo der Benutzer eine Meldung der Form *Möchten Sie das Dokument vor dem Schließen speichern?* erhält, deren Optionen es auch erlauben das Beenden zu verhindern.

Für alle Fälle soll direkt beim Empfang des Signals eine automatische Speicherung ausgelöst werden. Für den seltenen Sonderfall, dass sich die Animation in einem ungültigen

[3] Inzwischen ist der beschriebene Vorgang weitaus selektiver in der Auswahl der zu beendenden Prozesse

Zustand befindet, wird dennoch eine Speicherung erzwungen, dann allerdings unter abweichendem Dateinamen mit dem Zusatz *defective* sodass die Integrität der `backup.json` weiterhin gewahrt bleibt.

Nun kann es aber auch sein, dass der Benutzer gar keine Kontrolle mehr über die ***Record-UI*** hat, beispielsweise weil seine SSH-Verbindung zusammengebrochen ist. Damit das Programm weiterhin regulär über das Systemsignal SIGTERM beendet werden kann ohne dass ein SIGKILL erforderlich wird (was das automatische Speichern unmöglich machen würde) erhält der Benutzer eine Frist von derzeit 30 Sekunden, in denen er eine Eingabe tätigen muss, um das Beenden von außen zu unterbinden.

Über alle Vorgänge ist der Benutzer natürlich über das Terminal zu unterrichten.

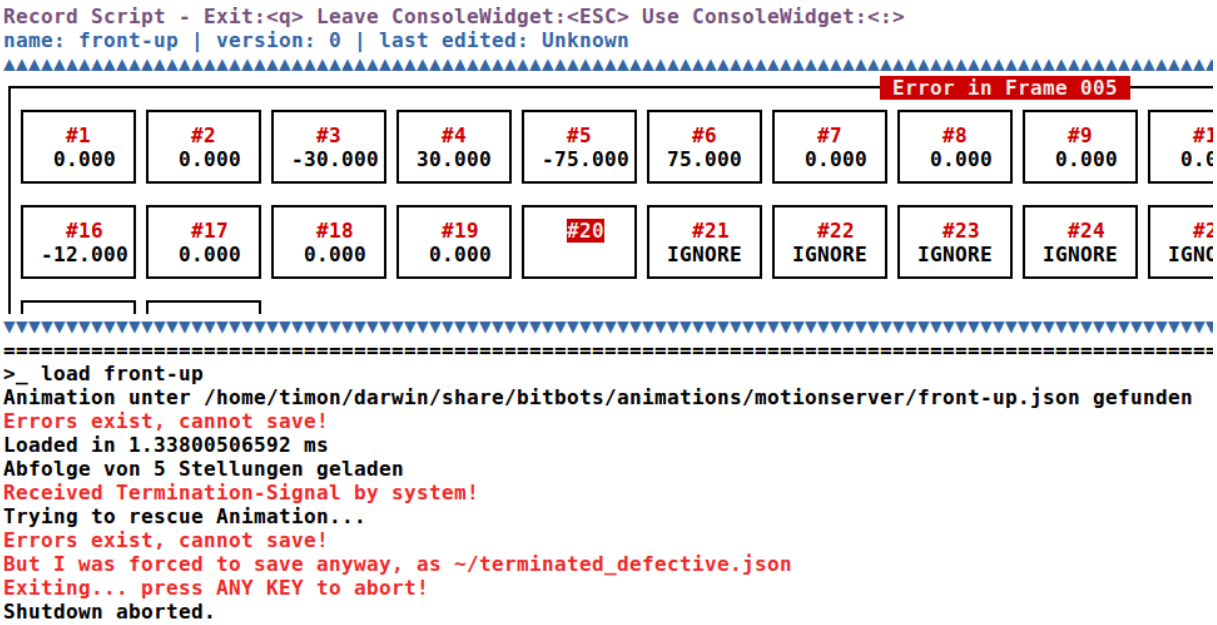


Abbildung 6.23.: Reaktion der ***Record-UI*** auf ein empfangenes SIGTERM

7. Bewusst ausgelassene Features

Obwohl die beschriebenen Features der ***Record-UI*** sehr vielfältig sind, gibt es unzählige weitere, die nicht eingeführt wurden. Teilweise geschah dies aus bewussten Überlegungen heraus, die in diesem Abschnitt wiedergegeben werden.

7.1. Keine interne Visualisierung der Animation

Hohberg [9, S.16/S.17] bennent die Visualisierung als ein wichtiges Feature für seinen *MotionEditor*:

> A visualisation of the robot is important for displaying the motion and to directly show how changes to motor values affect the robot's pose at a specific time. [...] Generally, the goal for the *MotionEditor* should be to allow the creation of motions without a real robot by visualizing the motion and the robot's movement in 3D.

Auch die Nao-Software *Choregraphe* (siehe Abschnitt 2.3) enthält von Haus aus eine solche Visualisierung [13], was von Hohberg [9, S.5] ausdrücklich als Vorzug hervorgehoben wird. Daher ist es notwendig, an dieser Stelle zu prüfen, ob eine Visualisierung der Animation eine notwendige Anforderung darstellt.

Vergangene Arbeiten im RoboCup haben gezeigt, dass sich das tatsächliche Verhalten von Robotern in der realen Welt nur zu einem gewissen Grad simulieren lässt. Nur weil ein Bewegungsablauf im Simulator funktioniert, bedeutet dies nicht, dass ein realer Roboter tatsächlich die nötige Stabilität aufweist.

Bei den *Hamburg Bit-Bots* ist der Ausruf *Im Simulator läuft's!* im Fehlerfall längst zu einem geflügelten Wort geworden, als humoristische Anspielung auf das Ergebnis früher Versuche, bei denen weite Teile der Software ausschließlich im Simulator getestet wurden.

Dem Nutzer eine Visualisierung des Bewegungsablaufes zu bieten, verleitet ihn dazu, eine große Anzahl an Keyframes der Animation zu erstellen, ohne zu testen, ob der Roboter

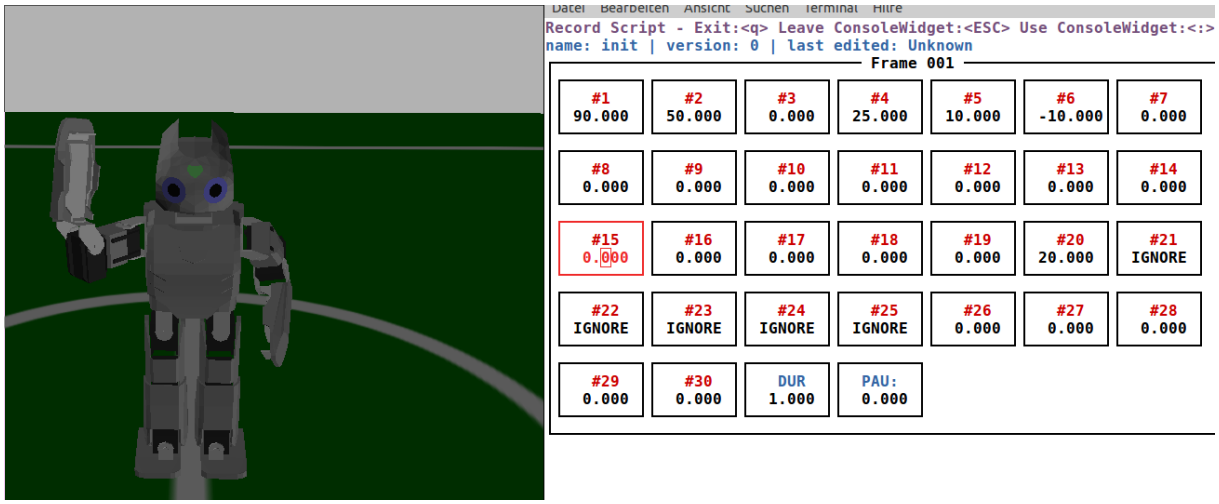


Abbildung 7.1.: ***Record-UI*** in Kombination mit der externen Visualisierung *glrobot*

tatsächlich in der Lage ist, die geplanten Keyframes zu erreichen. Eine Animationserstellung unter Benutzung eines realen Roboters ist daher in jedem Fall vorzuziehen.

Mit dem existierenden Framework der *Hamburg Bit-Bots* ist es allerdings dennoch ohne Weiteres möglich die Record-UI (wie jede andere für die Roboter geschriebene Software) in Kombination mit verschiedenen Visualisierungstools, bzw. Simulatoren zu nutzen, sodass auf Wunsch auch eine Visualisierung, bzw. in Abgrenzung zu [13] und [9] sogar eine direkte Simulation des physikalischen Roboterverhaltens möglich ist. In Abb. 7.1 sieht man beispielsweise die Visualisierung eines virtuellen Roboters mit den Werkzeugen *glrobot* und *Simulation* der *Hamburg Bit-Bots* ohne Einsatz eines echten Roboters und ohne simulation der Physik.

7.2. Keine Wahlmöglichkeit der Interpolation

Bereits im Abschnitt 2.2, in dem das Keyframeverfahren beschrieben wird, wird geschildert, dass eine Animation immer abhängig vom verwendeten Interpolationsalgorithmus ist. Diese Information wird auch zusammen mit den restlichen Animationsdaten gespeichert.

Bei Lektüre des Abschnitt 6 wird allerdings klar, dass nie eine Möglichkeit zum Interpolationswechsel in die ***Record-UI*** implementiert wurde. Tatsächlich wird grundsätzlich eine lineare Interpolation gewählt, wünscht der Benutzer einen anderen Interpolationsalgorithmus muss er nach wie vor die gespeicherte Animationsdatei von Hand editieren.

Die Wahlmöglichkeit des Interpolationsalgorithmus wäre in der Tat eine sinnvolle Ergänzung zum bisherigen Funktionsumfang. Allerdings hat sich in der Praxis gezeigt, dass die

Lineare Interpolation für die allermeisten Fälle ausreichend ist. Für *funktionelle* Animationen wie z.B. Aufstehen oder Ball Schießen wird überhaupt nur der lineare Animationsalgorithmus verwendet. *Sanfte* Bewegungen sind hier nicht von besonderem Interesse. Auch ist die lineare Interpolation für jeden Nutzer sehr einfach nachvollziehbar, während der Effekt anderer Interpolationsverfahren nicht sofort greifbar ist.

Für Animationen die primär der *Optik* dienen (z.B. Winken) ist allerdings tatsächlich zu empfehlen einen sanften Interpolationsalgorithmus zu verwenden, da er weniger mechanisch wirkt. Und auch für die *funktionellen* Animationen kann eine andere Interpolation ggf. Stabilitätsvorteile bringen, wenn beispielsweise aprupte Richtungswechsel ausgeglichen werden müssen. Bei alleiniger Verwendung der lineraren Interpolation wird häufig versucht mit Pausen und einer Verlängerung der Animatinonsdauer zwischen zwei Keyframes Stabilität zu erzielen, mit einer anderen Interpolation könnte hier ggf. der Gesamtablauf verkürzt und flüssiger gestaltet werden.

Daher wäre der Wechsel der Interpolation ein guter Schritt die ***Record-UI*** zu verbessern, auch wenn er nicht so zwingend notwendig ist wie Hohberg [9] impliziert.

8. Ergebnis

In diesem Abschnitt werden wir einige der durch die Entwicklungsschritte erreichten Eigenschaften der ***Record-UI*** ausführlicher diskutieren.

8.1. Arbeitsergonomie

Die ***Record-UI*** bietet mit den in Abschnitt 6 beschreibenen Änderungen eine ganze Reihe von Verbesserungen und Erleichterungen für den Benutzer. Die Bewegungsabläufe können in direkter Arbeit mit dem Roboter entworfen und verfeinert werden.

Es ist möglich gezielt *Posen* der Animation anzufahren und auch Teile der Animation auszuprobieren. Mit neuen Ideen kann gefahrlos experimentiert werden, da alle Aktionen reversibel sind. Mit dem unterstützenden Befehl `mirror` kann einfach und gezielt Symetrie in der Animation erzeugt werden.

Die beste Arbeitsweise ist es dabei mit zwei Personen zu arbeiten. Während die erste Person die ***Record-UI*** bedient, bringt die zweite Person den Roboter in die gewünschten Positionen und hat ein Auge darauf, dass die Hardware nicht beschädigt wird.

Aber auch die Arbeit mit einer Hand ist erheblich verbessert worden, vor allem durch die Autovervollständigungsfunktion ist die Bedienung mit einer Hand gegenüber dem *Record-Script* einfacher. Diese ist notwendig, da häufig der Roboter gestützt werden muss.

8.2. Nutzungsmöglichkeiten

Das fertige Produkt kann auf verschiedene Arten verwendet werden. Typisches Szenario ist es, mit einem Laptop neben einem Roboter zu sitzen, welcher über eine SSH-Verbindung, mit dem Laptop kommuniziert.

Es ist aber auch möglich die ***Record-UI*** ohne einen Roboter zu benutzen, entweder mit einer im Hintergrund laufenden Simulation, die einen realen Roboter imitiert, oder ohne eine solche. Wird kein Roboter simuliert, können einige Befehle in der ***Record-UI*** fehlschlagen, es ist aber weiterhin möglich Animationen zu editieren ohne das Ergebnis

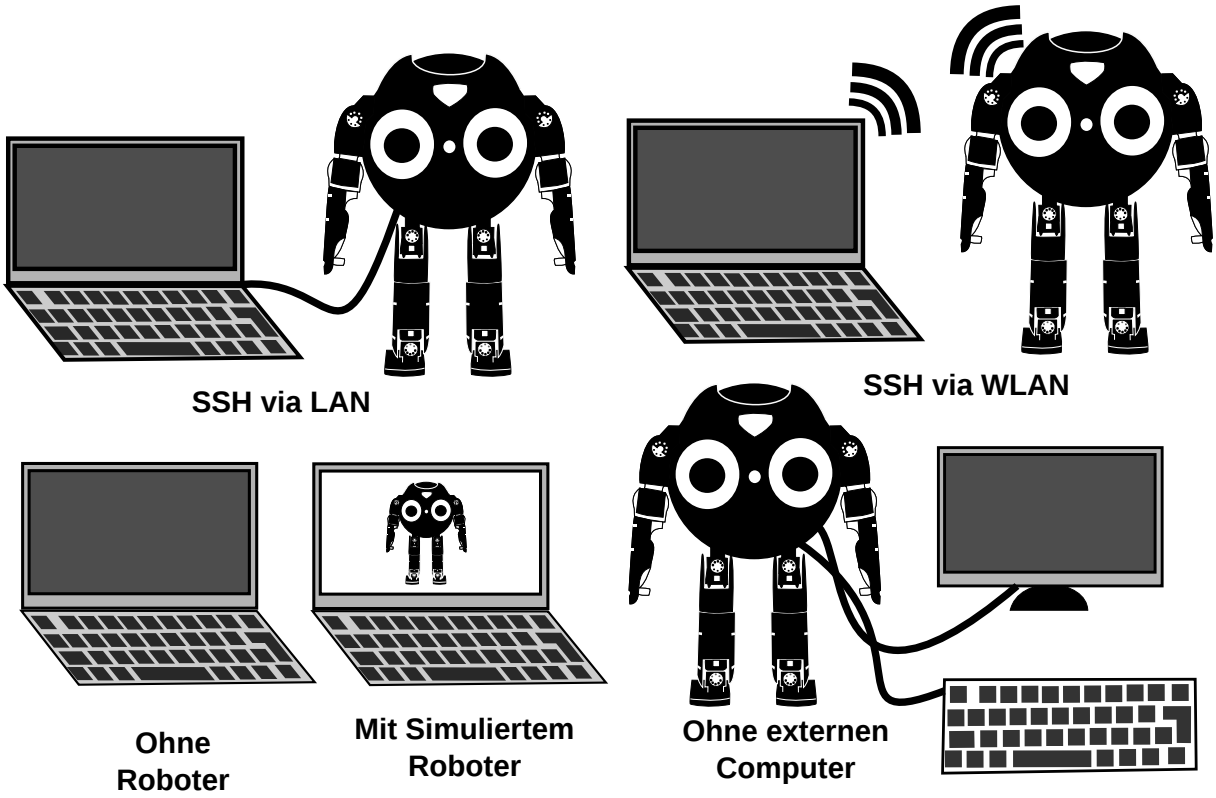


Abbildung 8.1.: Beispiele für Möglichkeiten die ***Record-UI*** zu nutzen

zu prüfen, beispielsweise wenn nur eine unkritische Anpassung vorgenommen werden soll. Dabei wird einfach ein lokales Terminal verwendet.

Eine besondere Möglichkeit besteht darin direkt den Roboter zu verwenden, da alle Roboter der *Hamburg Bit-Bots* einen vollwertigen Computer eingebaut haben und über die entsprechenden Anschlussmöglichkeiten verfügen, kann man auch direkt einen Bildschirm und eine Tastatur an den Roboter anschließen und die ***Record-UI*** ohne eine SSH-Verbindung nutzen.

Weder der Kanal SSH noch die Übertragung über LAN/WLAN ist ein zwingendes Kriterium. Wichtig ist nur, dass ein Terminal des die ***Record-UI*** ausführenden Systems zugänglich ist.

8.3. Unabhängigkeit von der verwendeten Plattform

Die ***Record-UI*** soll in der Robotikforschung Einsatz finden, daher ist klar, dass sich die zugrunde liegende Plattform ändern kann, beispielsweise durch veränderte Motorenanzahl oder -position. Alle benötigten Kenntnisse über den Aufbau des Roboters werden daher Konfigurationsdateien entnommen.

Derzeit existieren Konfigurationen für den Darwin-OP (20 Motoren), den Darwin-OP mit Angeschraubten Händen (24 Motoren), sowie für die neue Plattform der *Hamburg Bit-Bots* , den Hambot (24 Motoren).

Hohberg [9, S.20ff] beschreibt in diesem Zusammenhang das *Robot Description Format* (RDF) welches initial von Heinrich [8] definiert wurde. Ein Vergleich ist jedoch nicht Gegenstand dieser Arbeit. Es wurde entsprechend auch nicht geprüft, ob sich das RDF für den Einsatz in Kombination mit der ***Record-UI*** eignet.

8.4. Barrierefreiheit

Zum Zeitpunkt der Entwicklung gibt es bei den *Hamburg Bit-Bots* keine Nutzer, die einer Einschränkung unterliegen.

Da aber in der Informatik eher die geistigen Fähigkeiten gefordert sind, finden sich in der Gruppe potentieller Nutzer Personen, die verschiedenen körperlichen Einschränkungen unterliegen.

Daher lohnt sich eine kurze Betrachtung der entwickelten Software auch unter dem Gesichtspunkt der Barrierefreiheit.

8.4.1. Gehörlose und/oder stumme Nutzer

Die ***Record-UI*** Nutzt weder die Ein- noch Ausgabe von Sound und ist für gehörlose sowie stumme Nutzer uneingeschränkt verwendbar.

Die Roboter selbst geben unabhängig von der ***Record-UI*** Fehlermeldungen natürlichsprachlich über den Lautsprecher aus, dabei handelt es sich aber lediglich um einen zusätzlichen Kanal, die Einschränkung für gehörlose Nutzer dürfte unerheblich sein.

8.4.2. Nutzer mit eingeschränktem Sehvermögen

Bei vielen Arten von Einschränkungen ist es möglich die Optik der Nutzeroberfläche durch Anpassung des Farbthemas (monochrom, kontrastreich) an die Bedürfnisse des Nutzers anzupassen. Auch die Einstellungsmöglichkeiten des jeweils genutzen Terminals (Textgröße, Schriftart, evtl. auch Vergrößerung unter Nutzung von Scrollbalken) können die Record-UI solchen Nutzern zugänglich machen, die normalerweise Probleme bei der Betrachtung hätten.

Eine Benutzung durch blinde Personen ist nicht vorgesehen. Allerdings sollte es das Textbasierte Interface gegenüber einer *vollwertigen* GUI vereinfachen die Record-UI für die

Nutzung in Kombination mit einer Braille-Zeile zu erweitern. Auch für eine computergenerierte Sprachausgabe der Textausgabe, wäre nur eine sehr geringfügige Anpassung im Quellcode notwendig.

Da allerdings das gewählte Bedienkonzept auch in der Konsole graphische Elemente nutzt, müsste dafür vermutlich mit entsprechendem Aufwand eine Alternative geschaffen werden. Auch müsste der Informationsfluss neu strukturiert werden, damit genau die Ausgaben auf der Braille-Zeile erscheinen die eine entsprechende Präzision erfordern und damit die Sprachausgabe auf ein Minimum reduziert bleibt. Die ausführliche Diskussion würde leider den Rahmen dieser Arbeit sprengen.

8.4.3. Nutzer mit motorischen Einschränkungen

Die Nutzung der Maus ist vollkommen optional, die Record-UI erzwingt keine Eingaben unter Zeitvorgabe. Dadurch sollte jeder Benutzer der in der Lage ist eine Tastatur zu bedienen die Record-UI ohne gravierende Einschränkung nutzen können.

8.5. Sicherheit

8.5.1. Verletzungsrisiko für Mensch und Maschine

Bei der Arbeit mit Robotern ist grundsätzlich Vorsicht geboten, da immer ein Verletzungsrisiko durch die Motoren besteht. Sollte ein Bewegungsablauf nicht so verlaufen wie man es geplant hat, oder man aus Unachtsamkeit vom Roboter bei seiner Bewegung eingeklemmt werden, kann dies schmerzhafte Verletzungen und Sachschäden verursachen.

Auch mit Blick auf die in Zukunft immer größer und stärker werdenden Roboter im RoboCup sollte man sich bereits beim Softwaredesign unbedingt Gedanken über diesen Aspekt machen.

Bei den *Hamburg Bit-Bots* besteht ein wesentliches Verletzungsrisiko im Design des *Motionservers*. Die Implementation verfügt über eine Art *Kleinhirnfunktionalität*. Sollte ein Roboter sich in einer liegenden Position wiederfinden, so nimmt er an, dass er gefallen ist und führt automatisch einen Aufstehversuch durch. Für den Nutzer der Record-UI käme ein solches Verhalten völlig unerwartet, wenn er beispielsweise gerade den Roboter in eine liegende Position bringt, welche er als Ausgangsposition für seine Animation benötigt. Mögliche Folge wäre dabei ein schmerzhaftes Einklemmen beider Hände, was es zusätzlich erschwert den vorhandenen Not-Aus-Knopf am Roboter zu erreichen.

Damit dies nicht geschieht setzt die Record-UI bereits beim Start einen Block beim *Motionserver*, damit keine automatischen Bewegungen des Roboters möglich sind. Desweiteren

ist die Record-UI durchgehend so gestaltet, dass der Roboter nur auf explizite Nutzereingabe einen Bewegungsablauf startet. Ein weiteres Mittel zur Erhöhung der Sicherheit sind groß gewählte Standardwerte für die Ablaufdauer einer Bewegung. Eine Animation läuft so zunächst mit sehr geringer Geschwindigkeit, außer der Benutzer erhöht diese explizit.

8.5.2. Vertraulichkeit der Datenübertragung

Die Vertraulichkeit der Datenübertragung war zu keinem Zeitpunkt während der Entwicklung eine Überlegung. Durch die Verwendung von SSH ergibt es sich aber, dass die Kommunikation zwischen dem Roboter und dem verwendeten Laptop komplett Ende-zu-Ende verschlüsselt ist und, bei korrekter Konfiguration der SSH-Verbindung, als sehr sicher gelten kann.

Allerdings ist auch zu bemerken dass das Logging der ***Record-UI*** standardmäßig an das Debug-Framework der *Hamburg Bit-Bots* gekoppelt ist, welches wiederum vollständig unverschlüsselt arbeitet[1]. Diese Logging-Daten umfassen auch sämtliche Ein- und Ausgaben des Benutzers in die Konsole (siehe Abschnitt 5.5).

Sollte also beabsichtigt sein die von der ***Record-UI*** übertragenen Daten vertraulich zu halten, ist es erforderlich die Datenübertragung an das *Debug-Framework* zu unterbinden. Dafür genügt ein Eintrag in der Konfigurationsdatei.

8.5.3. Authentifizierung und Authorisierung

SSH übernimmt den Authentifizierungsprozess des Benutzters, und gemeinsam mit der für Linux-Systeme typischen Nutzer- und Rechteverwaltung kann sichergestellt werden, dass nur *befugte* Personen Kontrolle über den Roboter erlangen.

Während die Gefährdungslage im *RoboCup* sehr gering ist und die eingesetzten Roboter weitestgehten harmlos sind, ist es Wert anzumerken, dass sich die ***Record-UI*** dank SSH auch für Szenarien eignet in denen es auf hohe Sicherheitsanforderungen ankommt und bei denen es gilt unbedingt zu vermeiden, dass Unbefugte Kontrolle über den Roboter erlangen können.

8.6. Latenzbetrachtung

Bei der lokalen Nutzung der ***Record-UI*** ist keine Verzögerung zwischen Eingabe und Reaktion spürbar. Auch unter Nutzung einer SSH-Verbindung über ein LAN zum Roboter tritt eine Verzögerung höchst selten auf, nämlich genau dann wenn Datenpakete verloren

[1] Auf einem RoboCup-Wettbewerb kann auch der Datenverker der Roboter überwacht werden, um Betrugsfälle zu vermeiden.

gehen. Das SSH-Protokoll basiert auf dem TCP-Protokoll, welches Nachrichtenverlust bemerkt und den Versand der Pakete ggf. wiederholt.

Bei einer WLAN-Verbindung hingegen variiert die Qualität der Übertragung stark. Gerade bei starker Frequenzauslastung, wie sie auf RoboCup-Wettbewerben auftritt, ist mit Aussetzern zu rechnen, die mehrere Sekunden, in extremfällen Minuten dauern können. Auch komplette Verbindungsabbrüche sind in diesem Kontext nicht selten. Während eines Verbindungsaussetzers wird die ***Record-UI*** nicht mehr auf die Eingaben des Benutzers reagieren. Eine möglichst stabile Datenverbindung wird daher empfohlen.

Durch den Systemaufbau bleibt auch bei Verbindungsaussetzern der interne Systemzustand konstistent mit der UI, lediglich die Übertragung der UI an ein verbundenes Gerät kann sich verzögern. Probleme, die über das Aussetzen der Interaktion hinaus gehen, wurden in solchen Fällen bislang nicht beobachtet.

Auch bei einem völligen Verbindungszusammenbruch zum Roboter läuft die ***Record-UI*** auf dem Roboter weiter. Bei erneutem Aufbau einer Verbindung kann die Sitzung prinzipiell einfach fortgesetzt werden.

8.7. Mehrbenutzerfähigkeit

Eine interessante – tatsächlich eher zufällig entdeckte – Eigenschaft der ***Record-UI*** ist es, dass diese auch mit mehreren Endgeräten gleichzeitg genutzt werden kann.

Grund dafür ist der Einsatz des Konsolenfenstermanagers `screen` bei den *Hamburg Bit-Bots* , welcher es unter Linux ermöglicht mehrere Konsolensitzungen in einem Terminal zu verwalten. Dabei können sich auch mehrere Nutzer in die selbe Sitzung verbinden und teilen sich dort Ansicht und Cursor.

Es ist also möglich, dass sich zwei oder mehr Nutzer unter Verwendung ihres eigenen Endgeräts, Ansicht und Eingabe der selben Instanz der ***Record-UI*** teilen können. Dabei findet keine Konfliktbehandlung statt, alle Verbindungen sind völlig gleichberechtigt. Tätigen mehrere Nutzer beispielsweise gleichzeitig eine Texteingabe wird das in unbrauchbarem *Buchstabensalat* resultieren.

Eine sinnvolle Anwendung kann also darin bestehen, dass mehrere an der Erstellung einer Animation beteiligte Personen mit ihrem Laptop um den Roboter herum sitzen und in Absprache miteinander jeweils eine Person die aktive Steuerung der ***Record-UI*** übernimmt, wobei sie ohne Platzwechsel ihr eigenes Gerät nutzen kann[2].

[2]Dabei ist, technisch bedingt, allerdings allen Nutzern die selbe Konfiguration, z.B. des Farbschemas aufgezwungen, da es sich um die selbe Instanz der ***Record-UI*** handelt

8.8. Auswertung Softwaretests

Die in Abschnitt 5.1 beschriebenen Softwaretests haben sich im Verlauf der Entwicklung als extrem nützlich erwiesen. Viele Fehler konnten früh erkannt werden und der Gesamtentwicklungsverlauf hat sich dadurch beschleunigt.

Als mangelhaft kann allerdings die Testabdeckung im Hinblick auf die UI-Komponenten betrachtet werden. Da es schwierig war die Eingaben in die Keyframekomponenten zu prüfen wurde darauf verzichtet. Hier hatten sich später noch Fehler eingeschlichen, die von Hand ermittelt werden mussten. Im Nachhinein betrachtet wäre es sinnvoll gewesen für die UI spezifische Komponententests einzuführen, welche auch die Eingaben in die entspechenden Formularfelder separat prüfen.

Wesentlich zur Testqualität beigetragen hat es, das vorhandene Simulationsframework der *Hamburg Bit-Bots* automatisch während der Tests im Hintergrund laufen zu lassen. So konnten auch die Befehle getestet werden, die außerhalb eines konkreten Roboters fehlschlagen würden.

```
Name                    Stmts   Miss  Cover
-----------------------------------------
record/commands.py        382    126    67%
record/handler.py          30      3    90%
record/help_parser.py      84     11    87%
record/record.py          367     65    82%
record/ui.py              255     80    69%
record/util.py            197     68    65%
record/widget.py          537    259    52%
-----------------------------------------
TOTAL                    1852    612    67%
```

Abbildung 8.2.: Testabdeckung der Record-UI
laut Coverage.py version 4.0a5

9. Fazit

Es wurde erfolgreich die Anwendung ***Record-UI*** geschaffen, welche in einer beliebigen Terminalumgebung, egal ob lokal oder *remote*, nutzbar ist und dabei pseudo-graphische Elemente nutzt, um die Nutzerfreundlichkeit zu erhöhen.

Der Vergleich lässt sich dabei in erster Linie mit der Ausgangssituation ziehen, gegenüber der signifikante Verbesserungen des Oberflächenaufbaus und der Funktionalität, mit Mehrwert sowohl für Erfahrene als auch Unerfahrene Nutzer erzielt werden konnten. Immer wieder verwiesen wird auch auf die Arbeit von Hohberg [9], welcher einen alternativen Ansatz gewählt hat.

In der Arbeit wurde darüber hinaus demonstriert, wie auch heute noch mit vergleichsweise modernen Technologien kommandozeilenbasierte Programme gestaltet werden können und welche Vor- und Nachteile sich daraus ergeben. Neben der Nutzerfreundlichkeit konnten auch einige wichtige Fragen zum Bereich Sicherheit und Barrierefreiheit positiv beantwortet werden.

Die ***Record-UI*** ist dabei mit Blick auf moderne Standards der Softwaretechnik entworfen worden, sie kann auch nach Abschluss dieser Arbeit gepflegt und weiterentwickelt werden. Überdies können einzelne Elemente bei Bedarf entkoppelt und in anderen Zusammenhängen genutzt werden. Sie bildet damit eine wichtige technische Grundlage für weitere Forschung im Rahmen der RoboCup-AG *Hamburg Bit-Bots* .

9.1. Ausblick und künftige Forschung

9.1.1. Display am Roboter

Die neueste Robotergeneration der *Hamburg Bit-Bots* , mit dem Plattformnamen *Hambot*, soll ein kleines Monochromdisplay auf der Rückseite erhalten. Für die Anzeige der vollwertigen ***Record-UI*** wird diese Fläche ziemlich sicher zu klein sein. Eventuell lässt es sich aber für zusätzliche Statusinformationen auch durch die ***Record-UI*** nutzen.

Das Display soll *Touch*, also Berührungseingabe unterstützen. Ebenfalls interessant wäre daher die Frage, ob sich die Record-UI anpassen lässt, um diese Fläche für zusätzliche Nutzereingaben verwendbar zu machen.

9.1.2. Natürlichsprachliche Befehlseingabe

Für die aktuelle Version der ***Record-UI*** wird empfohlen mit zwei Personen daran zu arbeiten. Bei der Positionierung des Roboters und dessen Absicherung gegen Sturz kann man kaum genug Hände zur Verfügung haben und ein Paar Hände wird benötigt, um die ***Record-UI*** optimal zu bedienen.

Um als Einzelperson optimal eine Animationserstellung durchzuführen, müsste man seine Eingaben tätigen können ohne die Hände zu benutzen. Die Anwendung würde daher von einer zuverlässigen, natürlichsprachlichen Eingabe wirklich profitieren.

Während es nicht ohne großen zusätzliche Aufwand möglich wäre ein Mikrophon am Laptop dafür zu verwenden, besitzen alle derzeit verwendeten Robotermodelle bereits von Haus aus ein oder mehrere Mikrophone (teilweise in die Kamera integriert). Paetzel [12] hat in diesem Zusammenhang bereits Möglichkeiten der Spieler-Spieler Kommunikation erforscht. Erste Untersuchungen zur Möglichkeit der Mensch-Roboter Kommunikation und der natürlichsprachlichen Roboter-Roboter Kommunikation lassen vermuten, dass es während eines Wettkampfes zu laut und die natürlichsprachliche Kommunikation zu Fehleranfällig für eine Kommunikation über größere Distanzen ist.

Während der Arbeit mit der ***Record-UI*** befindet man sich aber in unmittelbarer Nähe zum Roboter und die Umgebung düfte meist ruhiger ausfallen als während eines laufenden Spiels. Eine Untersuchung, ob die Mikrophone der Roboter zur Eingabe bei Verwendung der ***Record-UI*** geeignet sind, wäre daher höchst interessant.

A. Literaturverzeichnis

[1] Marco Antonelli, Fabio Dalla Libera, Emanuele Menegatti, Takashi Minato, and Hiroshi Ishiguro. Intuitive Humanoid Motion Generation Joining User-Defined Key-Frames and Automatic Learning. In Luca Iocchi, Hitoshi Matsubara, Alfredo Weitzenfeld, and Changjiu Zhou, editors, *RoboCup 2008: Robot Soccer World Cup XII*, volume 5399 of *Lecture Notes in Computer Science*, pages 13–24. Springer Berlin Heidelberg, 2009. ISBN 978-3-642-02920-2. doi: 10.1007/978-3-642-02921-9_2. URL `http://dx.doi.org/10.1007/978-3-642-02921-9_2`.

[2] Daniel J Barrett and Richard E Silverman. *SSH, the Secure Shell: the definitive guide*. O'Reilly Media, Inc., 2001.

[3] Oren Ben-Kiki, Clark Evans, and Ingy döt Net. YAML Ain't Markup Language (YAML™) Version 1.2, October 2009. URL `http://www.yaml.org/spec/1.2/spec.html`. Letzter Zugriff: 2015-08-22.

[4] Marc Bestmann, Bente Reichardt, and Florens Wasserfall. Hambot: An Open Source Robot for RoboCup Soccer. Technical report, Universität Hamburg, 2015. URL `http://robocup.informatik.uni-hamburg.de/wp-content/uploads/2012/05/Hambot.pdf`.

[5] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, RFC Editor, March 2014. URL `http://www.rfc-editor.org/rfc/rfc7159.txt`. `http://www.rfc-editor.org/rfc/rfc7159.txt`.

[6] Stefan Czarnetzki, Sören Kerner, and Daniel Klagges. Combining key frame based motion design with controlled movement execution. In Jacky Baltes, Michail G. Lagoudakis, Tadashi Naruse, and SaeedShiry Ghidary, editors, *RoboCup 2009: Robot Soccer World Cup XIII*, volume 5949 of *Lecture Notes in Computer Science*, pages 58–68. Springer Berlin Heidelberg, 2010. ISBN 978-3-642-11875-3. doi: 10.1007/978-3-642-11876-0_6. URL `http://dx.doi.org/10.1007/978-3-642-11876-0_6`.

[7] DIN EN ISO 9241-110:2006. Ergonomie der Mensch-System-Interaktion - Teil 110: Grundsätze der Dialoggestaltung; Deutsche Fassung, September 2006.

[8] Steffen Heinrich. Development of a Multi-Level Sensor Emulator for Humanoid Robots. Master's thesis, Freie Universität Berlin, Januar 2012.

[9] Simon Philipp Hohberg. Interactive Key Frame Motion Editor for Humanoid Robots. Bachelorarbeit, Freie Universität Berlin, Februar 2012. URL https://maserati.mi.fu-berlin.de/fumanoids/wp-content/papercite-data/pdf/hohberg2012.pdf.

[10] Vstone Co. ltd. RobovieMaker2 User Guide, April 2008. URL http://kumotek.com/download/manual/RM2manual2_eng.pdf. Letzter Zugriff: 2015-08-29.

[11] Mark Lutz, David Ascher, and Dinu C Gherman. *Einführung in Python*. O'Reilly Germany, 2007.

[12] Maike Paetzel. Spielerkoordination in RoboCup-Fußballspielen mittels gesprochener Sprache. Bachelorarbeit, Universität Hamburg, Juni 2013. URL http://robocup.informatik.uni-hamburg.de/wp-content/uploads/2015/02/BachelorThesis_MaikePaetzel.pdf.

[13] Aldebaran Robotics. Choregraphe User Guide. URL http://doc.aldebaran.com/1-14/software/choregraphe/index.html. Letzter Zugriff: 2015-04-29.

[14] Robotis. ROBOTIS-OP (a.k.a. DARWIN-OP) project, November 2010. URL http://sourceforge.net/projects/darwinop/. Repository aller Darwin-OP Resourcen. Letzter Zugriff: 2015-08-29.

[15] Technical Commitee, RoboCup Humanoid. RoboCup Soccer Humanoid League Rules and Setup For the 2015 Competition in Hefei, Juni 2015. URL http://www.robocuphumanoid.org/wp-content/uploads/HumanoidLeagueRules2015-06-29.pdf.

[16] Leila Tkayama and Eser Kandogan. Trust as an Underlying Factor of System Administrator Interface Choice, April 2006.

[17] Guido van Rossum, Barry Warsaw, and Coghlan Nick. PEP0008 – Style Guide for Python Code, Juli 2001. URL https://www.python.org/dev/peps/pep-0008/.

[18] Ian Ward et al. Urwid Reference. URL http://urwid.org/. Letzter Zugriff: 07-07-2015.

[19] T. Ylonen and C. Lonvick. The Secure Shell (SSH) Connection Protocol. RFC 4254, RFC Editor, January 2006. URL http://www.rfc-editor.org/rfc/rfc4254.txt. http://www.rfc-editor.org/rfc/rfc4254.txt.

B. Anhang

B.1. Auszug aus einem Test-Case

```
    def test_record(self):
        reason = 'record with too many arguments should return False'
        test = self.commander.run('record', [1, 2, 3])
        self.assertFalse(test, reason)

        reason = 'record with invalid argument should return False'
        test = self.commander.run('record', ['nothing'])
        self.assertFalse(test, reason)

        reason = 'recorder should not have a keyframe after failed record'
        test = self.recorder.anim
        self.assertFalse(test, reason)
```

B.2. Auszug aus der Datei joints.yaml

```
- id: 1
  limits: {default: 0, max: 180, min: -180}
  name: RShoulderPitch
  tags: [arms, rarm, shoulders, right, '1']
  opposing: 2
  inverted: True
- id: 2
  limits: {default: 0, max: 180, min: -180}
  name: LShoulderPitch
  tags: [arms, larm, shoulders, left, '2']
  opposing: 1
  inverted: True
```

B.3. Eine Animationsdatei

```
{
    "default_interpolator": "CatmullRomInterpolator",
    "name": "init",
    "keyframes": [
        {
            "duration": 1.0,
            "pause": 0.0,
            "goals": {
                "LShoulderPitch": 50,
                "LShoulderRoll": 25.0,
                "RShoulderRoll": -25.0,
                "..."
            }
        }
    ]
}
```

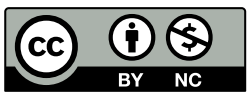
C. Abbildungsverzeichnis

Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Studiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel – insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen – benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Ort / Datum: .....

(Unterschrift)



Insbesondere stimme ich der Veröffentlichung in Bibliotheken zu.