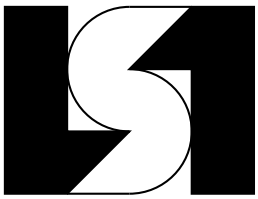
Diplomarbeit

Entwicklung und Visualisierung eines virtuellen astronomischen Instruments

Kai Lars Polsterer



Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund

14.03.2003

Betreuer:

Prof. Dr.-Ing. Claudio Moraga
Prof. Dr. rer. nat. Ralf Jürgen Dettmar

Dieses Dokument wurde mit LaTeX 2ε erstellt, einem System zur Erstellung von Texten, das 1985 von LESLIE LAMPORT entworfen wurde (siehe [Kop00]).

Es basiert auf der Textsatz-Sprache TeX, die von DONALD E. KNUTH 1984 für das Buch [Knu97], entwickelt wurde.

Das Literaturverzeichnis wurde mit BibTeX erstellt.

Als Editor wurde LyX 1.1.6fix4 verwendet.

Inhaltsverzeichnis

1 Einleitung **7**
- 1.1 Large Binocular Telescope . . . 7
- 1.2 Das Lucifer Projekt . . . 9
- 1.3 Notwendigkeit / Aufgaben eines virtuellen Instruments . . . 11

2 Technischer Hintergrund **13**
- 2.1 Lucifer . . . 13
  - 2.1.1 Elektronik . . . 13
  - 2.1.2 Computerschnittstelle . . . 14
- 2.2 Mechanik . . . 14
  - 2.2.1 Schrittmotoren . . . 14
  - 2.2.2 Referenzschalter . . . 17
  - 2.2.3 Rasten . . . 17

3 Theoretische Vorüberlegungen **19**
- 3.1 Simulation der Mechanik . . . 19
  - 3.1.1 Definition der zu simulierenden Instrumentenparameter . . . 19
  - 3.1.2 Anforderungen an die Simulation der Mechanik . . . 20
  - 3.1.3 Gewählter Simulationsansatz . . . 21
- 3.2 Emulation der Elektronik . . . 23
  - 3.2.1 Nachbildung der Steuerelektronik . . . 24
  - 3.2.2 Kommunikation . . . 28
- 3.3 Visualisierung . . . 29
  - 3.3.1 Anforderungen an die Visualisierung . . . 29
  - 3.3.2 Anforderungen an eine graphische Benutzungsoberfläche . . . 31
  - 3.3.3 Gewählte Visualisierung und Strukturierung . . . 32

4 Umsetzung der Theorie **35**
- 4.1 Java . . . 35
  - 4.1.1 Eigenschaften von Java . . . 36
  - 4.1.2 Gründe für die Wahl von Java . . . 38
- 4.2 Entwicklungsumgebung . . . 39
  - 4.2.1 Gründe für die Wahl von Together . . . 40
  - 4.2.2 Benutzungsoberfläche . . . 41
  - 4.2.3 Durch Together erstellte Dokumentation . . . 41
- 4.3 Simulation der Mechanik . . . 42
  - 4.3.1 Datenstruktur der Mechanik . . . 43
  - 4.3.2 Zeit- und Zufallserzeugung . . . 44
  - 4.3.3 Winkelsimulation . . . 45
- 4.4 Emulation der Elektronik . . . 47

4.4.1 Ansteuerung der Simulation 48
4.4.2 Netzwerkanbindung 51
4.4.3 Ausgabe zeitversetzter Nachrichten 51
4.4.4 Ausführen der Kommandos 52
4.5 Visualisierung 54
4.5.1 Visualisierung der Kommunikationsdaten 55
4.5.2 Visualisierung der Mechanik 56
4.5.3 Benutzungsoberfläche 68

5 Lucifer VR **69**
5.1 Hardwareanforderung und Installation 69
5.2 Benutzungsoberfläche 70
5.2.1 Strukturansicht 70
5.2.2 Kontextansicht 71
5.2.3 Geometrische Ansicht 75
5.2.4 Kommunikationsdaten 77
5.3 Ergebnisse 79

6 Zusammenfassung und Ausblick **81**

A Glossar **83**

B Symbolindex **85**

C Steuerkommandos **87**

D Danksagungen **93**

Das tiefste und erhabenste Gefühl, dessen wir fähig sind, ist das Erlebnis des Mystischen. Aus ihm allein keimt wahre Wissenschaft. Wem dieses Gefühl fremd ist, wer sich nicht mehr wundern und in Ehrfurcht verlieren kann, der ist seelisch bereits tot.

Albert Einstein

Vorwort

Die Astronomie hat seit Menschengedenken die Menschheit in ihren Bann gezogen. Schon als Kind interessierte ich mich für die Astronomie. Mit den Jahren wurde aus diesem Interesse eine ernsthafte Beschäftigung. Jetzt, mit Abschluss meines Studiums, konnte ich in meiner Diplomarbeit die Astronomie mit der Informatik vereinen. Die letzten sechs Monate waren, obwohl zwischenzeitlich sehr anstrengend, die schönste Zeit meines Studiums. Das während der letzten Jahre gesammelte Wissen konnte endlich, noch dazu in der mir sehr am Herzen liegenden Astronomie, angewandt und vertieft werden.

Ich hoffe diese Diplomarbeit wird dem Leser nicht nur den Inhalt der Informatik vermitteln. Vielmehr würde es mich freuen, wenn das Intresse für die Astronomie, sofern es nicht schon existiert, geweckt wird.

Inhaltsübersicht

In dieser Diplomarbeit wird die Entwicklung sowie die Visualisierung eines virtuellen astronomischen Instruments behandelt. Im Rahmen der Arbeit wurde exemplarisch das Lucifer Instrument als Grundlage gewählt. Lucifer (***l****bt near infrared spectroscopic* ***u****tility with* ***c****amera and* ***i****ntegral* ***f****ield unit for* ***e****xtragalactic* ***r****esearch*) ist eines der Instrumente für ein in Arizona neu errichtetes Großteleskop, das *„Large Binocular Telescope“* (LBT). Lucifer soll im nahinfraroten Wellenlängenbereich zur Bildgewinnung und spektrographischen Analyse von extragalaktischen Objekten genutzt werden. Aufgrund des beobachteten Wellenlängenbereichs muss Lucifer kryogenisch auf 80°K gekühlt werden.

Nach einer kurzen Einleitung, in der sowohl das LBT als auch das Lucifer Projekt vorgestellt werden, folgen zunächst einige technische Grundlagen. Dabei werden neben der für die Steuerung benötigten Elektronik und ihrer Computerschnittstelle auch grundlegende mechanische Elemente beschrieben.

In einem theoretischen Kapitel werden Vorüberlegungen zur Realisierung eines virtuellen astronomischen Instruments angestellt. Dazu zählt der Entwurf eines entsprechenden Simulationsmodells sowie die Überlegungen zum Entwurf eines Kommandoparsers. Die Aufgabe dieses Kommandoparsers besteht darin die Steuerbefehle zu entschlüsseln. Außerdem wird eine Anforderungsanalyse für eine entsprechende Instrumentenvisualisierung angefertigt.

In dem darauf folgenden Kapitel wird die Umsetzung dieser Vorüberlegungen näher beschrieben. Dieser Teil der Diplomarbeit ist eher praktisch orientiert und behandelt die Implementierung des virtuellen astronomischen Instruments mittels der objektorientierten Programmiersprache Java. Dabei wird UML, eine universell einsetzbar Modellierungsmethodik, verwendet um Sachverhalte genauer zu beschreiben. Zusätzlich wird in diesem Kapitel die verwendete Entwicklungsumgebung vorgestellt.

Zuletzt wird der im Rahmen dieser Diplomarbeit entstandene Softwareprototyp vorgestellt. Außerdem wird der Prototyp mit einer real existierenden Testeinheit des Max Planck Instituts für Astronomie verglichen.

Je mehr man schon weiß, desto mehr hat man noch zu lernen. Mit dem Wissen nimmt das Nichtwissen in gleichem Grade zu oder vielmehr das Wissen des Nichtwissens.

Friedrich Schlegel

Kapitel 1

Einleitung

Ziel dieser Diplomarbeit ist es, Methoden zu entwickeln und umzusetzen, die es ermöglichen, ein virtuelles astronomisches Instrument zu erstellen. Dieses hat essentielle Bedeutung für die Entwicklung der Steuersoftware (siehe Unterkapitel 1.3). Die entwickelten Methoden sowie deren Umsetzung sollen in den folgenden Kapiteln beschrieben werden. Für diese Diplomarbeit wurde das Lucifer Instrument, das eines der *„first light“*[1] Instrumente des *„Large Binocular Telescope“* (LBT[2]) sein wird, als Grundlage gewählt. Es ist jedoch möglich, die in Zusammenhang mit dieser Diplomarbeit gewonnenen Erkenntnisse auch auf andere astronomische Instrumente anzuwenden.

Zunächst werden das LBT sowie das Lucifer Projekt vorgestellt. Dies soll einen Überblick über das astronomische Projekt, in das diese Diplomarbeit eingebettet ist, geben. Außerdem werden die Anforderungen an ein virtuelles astronomisches Instrument aufgeführt. In Kapitel 2 wird das notwendige Wissen über die nachzubildende Technik vermittelt. Kapitel 3 enthält die theoretischen Vorüberlegungen zur Realisierung eines virtuellen Instruments. Dabei wird zwischen der Simulation der Mechanik, der Nachbildung der Steuerelektronik und der Visualisierung des virtuellen astronomischen Instruments unterschieden. Der Realisierung ist das darauf folgende Kapitel 4 gewidmet. In ihm wird beschrieben wie die drei genannten Gebiete umgesetzt wurden. Im Anschluss wird in Kapitel 5 der im Rahmen dieser Diplomarbeit erstellte Softwareprototyp vorgestellt. Zum Ende dieser Diplomarbeit findet sich eine kurze Zusammenfassung sowie ein Ausblick über mögliche weitere Arbeiten, die an dem virtuellen astronomischen Instrument noch vorgenommen werden könnten.

1.1 Large Binocular Telescope

Bevor das Lucifer Projekt beschrieben wird, ist es wichtig das dazugehörige Großteleskop zu betrachten. Das LBT Projekt ist eine Kooperation verschiedener Institute in Arizona (USA), Ohio (USA), Italien und Deutschland. Dabei tragen Arizona, Italien sowie Deutschland jeweils 25% der Baukosten von ca. 100 Mio US $. Die restlichen 25% verteilen sich zu gleichen Teilen auf den Bundesstaat Ohio sowie eine extra gegründete Forschungsgesellschaft. Der deutsche Anteil wird dabei durch die LBT Beteiligungsgesellschaft finanziert, die sich aus dem Max Planck Institut für Astronomie in Heidelberg (MPIA), der Landessternwarte Heidelberg (LSW), dem Max Planck Institut für Radioastronomie in Bonn (MPIfR), dem Max Planck Institut für extraterrestrische Physik in München (MPE) sowie dem astrophysikalischem Institut in Potsdam (AIP) zusammensetzt. Entsprechend der finanziellen Beteiligung wird später den Instituten Beobachtungszeit an dem Teleskop zur Verfügung gestellt.

[1] *„first-light“* bezeichnet die erste Nutzung eines Teleskops

[2] Ein die Abkürzungen enthaltendes Glossar befindet sich in Anhang A.

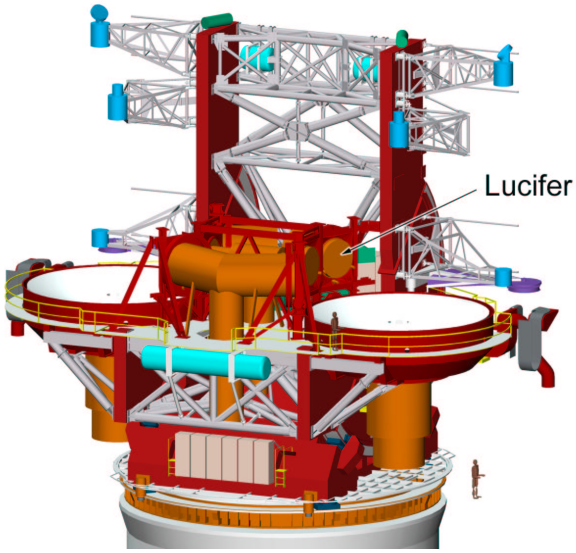


Abbildung 1.1: Skizze des LBT (Quelle: EIE)

Das LBT wird zwei Primärspiegel für die Datengewinnung nutzen (siehe Abb. 1.1). Diese sind bereits von dem Steward Observatory Mirror Lab, das der Universität von Arizona angehört, mit einer gewichtssparenden Wabenstruktur gefertigt worden. Jeder dieser rotierend gegossenen Spiegel wiegt 16 Tonnen, hat einen Durchmesser von 8,4 Metern und eine Brennweite von 9,6 Metern. Durch ein spezielles Lüftungssystem ist es möglich, die Spiegel innerhalb von 45 Minuten an die Umgebungstemperatur anzupassen. Dies ist nötig, um störende Einflüsse, die durch Luftturbulenzen oberhalb der Spiegeloberfläche entstehen, zu minimieren. Die Oberfläche beider Spiegel zusammen beträgt $110m^2$, was einem einzelnen Spiegel mit 11.8 Metern Durchmesser entspricht. Dadurch, dass die Spiegel in einem Abstand von 14.4 Metern (Mittelpunkt zu Mittelpunkt) befestigt sind, entspricht das Auflösungsvermögen beider Spiegel, wenn ihre Strahlengänge interferometrisch zusammengefasst werden, der eines 22.8 Meter durchmessenden Spiegels. Damit aufgrund der Erdgravitation und Leichtbauweise hervorgerufene Spiegelverformungen ausgeglichen werden können, sind beide Primärspiegel aktiv gelagert.

Neben dieser aktiven Optik verfügt das LBT über eine adaptive Optik. Diese verformt den Sekundärspiegel, welcher oberhalb des Primärspiegels montiert ist (siehe Abb. 1.1). Dadurch werden die durch die Atmosphäre der Erde hervorgerufenen Störungen ausgeglichen. Dazu ist der 91cm durchmessende und 1.6mm dicke Sekundärspiegel auf 918 aktiven Elementen gelagert. Für die Verformungsberechnung des Sekundärspiegels wird entweder die Verformung von Referenzsternen beobachtet, oder mittels eines Sodium-Lasers - der in den höheren Luftschichten eine visuell sichtbare Reaktion hervorruft - eine Referenzquelle erzeugt und analysiert. Durch den Einsatz einer adaptiven Optik steht das Teleskop für eine beugungsbegrenzten, sowie einen durch die Atmosphäre limitierten Beobachtungsmodus zur Verfügung.

Über den Tertiärspiegel wird das Licht in das Lucifer Instrument gelenkt (siehe Abb. 1.1). Das entsprechende zweite Lucifer Instrument befindet sich an der gleichen Stelle des anderen Primärspiegels. Neben dem von [Mea00] beschriebenen Nahinfrarot-Spektrographen soll ein zweites Paar im optischen und ultravioletten Wellenlängenbereich arbeitender Spektrographen verwendet werden (siehe [Oea00]).

Das ausführliche Design der Teleskopstruktur wurde von European Industrial Engineering (EIE) in Italien angefertigt (Abb. 1.1). Die Gesamtmasse der beweglichen Teleskopteile wird ca. 600 Tonnen erreichen. Um eine möglichst ruhige und ruckfreie Bewegung zu ermöglichen, schwimmt das ganze Teleskop auf speziellen Ölkissen. Jede der beiden Achsen wird von je vier Motoren angetrieben.

Abbildung 1.2: LBT Gebäude

Errichtet wird das LBT in 3191 Metern Höhe auf dem Mount Graham in Arizona. Die Abb. 1.2 zeigt das bereits fertiggestellte Teleskopgebäude. Damit oberhalb der Baumwipfel auftretende Luftturbulenzen die Aufnahmequalität des Teleskops nicht beeinträchtigen, wurde unterhalb des eigentlichen Teleskopbereiches ein ca. 20 Meter hoher Gebäudeteil errichtet. In diesem befinden sich der Kontrollraum, einige Büros, Lagerräume sowie die Wohnräume für die Astronomen und des zur Nutzung notwendigen Personals.

In dem oberen drehbaren Gebäudeteil befindet sich das Teleskop. Während der Teleskopnutzung teilt sich dieser in zwei Teile, die auseinanderschwenken und so dem Teleskop den Himmel freigeben. Durch eine spezielle Isolierung sowie ein ausgeklügeltes Lüftungssystem wird das Teleskop in kürzester Zeit an die Außentemperatur angepasst. Dies ist nötig, um störende, aus Temperaturunterschieden resultierende Luftunruhe zu minimieren.

Aktuelle Bilder zu dem Stand des LBT Projekts finden sich unter `medusa.as.arizona.edu/lbtwww/`, woher auch alle Informationen zu diesem Unterkapitel entnommen sind.

1.2 Das Lucifer Projekt

Lucifer ist ein Akronym, gebildet aus der englischen Instrumentenbezeichnung ***„l**bt nir spectroscopic **u**tility with **c**amera and **i**ntegral- **f**ield unit for **e**xtragalactic **r**esearch“*. Hierbei handelt es sich um ein Projekt, das in Zusammenarbeit von fünf deutschen Instituten durchgeführt wird. Diese sind die Landessternwarte Heidelberg, das Max Planck Institut für Astronomie in Heidelberg, das Max Planck Institut für extraterrestrische Physik in Garching, die Fachhochschule für Technik und Gestaltung in Mannheim (FHTG) sowie das astronomische Institut der Ruhr-Universität Bochum (AIRUB).

Ziel des Projekts ist es, eines der *„first-light“* Instrumente des LBT zu entwickeln. Das LBT wird in seiner endgültigen Ausbauphase mit zwei identischen Lucifer Instrumenten - für jeden der zwei Spiegel jeweils einem - ausgestattet werden.

Da das Lucifer Instrument im nahinfraroten Wellenlängenbereich zwischen 900nm und 2500nm zum Einsatz kommt, ist es notwendig das Instrument auf 80°K kryogenisch zu kühlen,

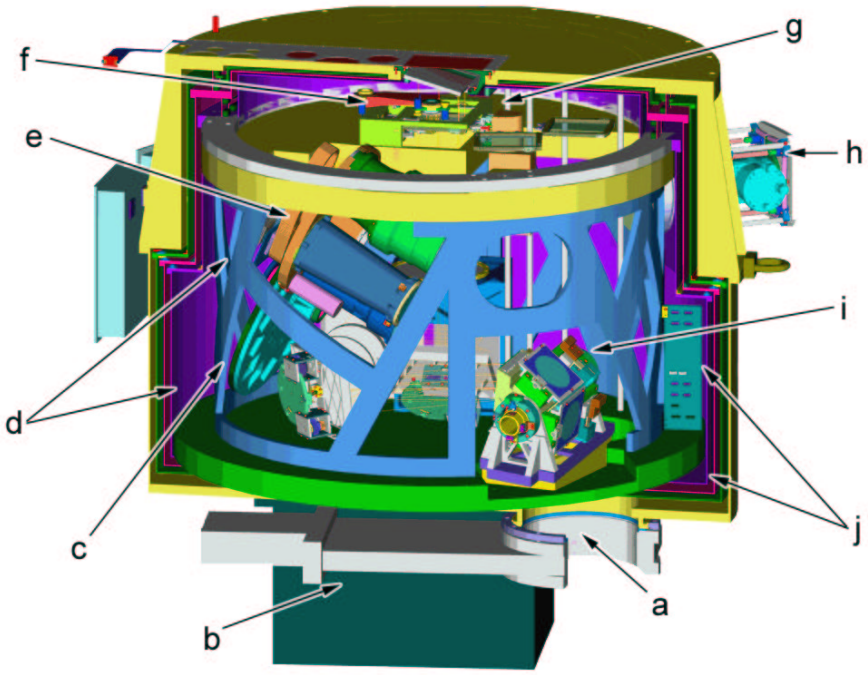


Abbildung 1.3: Lucifer Instrument (Quelle: LSW)

um Störeinflüsse thermischer Art zu minimieren. Das Instrument verfügt über drei automatisch auswechselbare Optiken und kann somit das Bildfeld dem jeweiligen Beobachtungsmodus des Teleskops anpassen.

Neben der Nutzung des Lucifer Instruments für die normale Bildgewinnung kann es auch Daten als Spektrograph sammeln. Durch die Möglichkeit zwei Gitter von unterschiedlicher Auflösung zu benutzen, wird das Instrument in die Lage versetzt, das einfallende Licht seiner Wellenlänge entsprechend zu zerlegen. Dabei kann die Ausrichtung der Gitter mit einer Winkelgenauigkeit im Sekundenbereich gesteuert werden. Durch den Einsatz spezieller lasergefertigter Masken ist es möglich den zu spektroskopierenden Bereich zu selektieren. Damit stellt das Lucifer Instrument das erste astronomische Instrument dar, in dem sowohl normale Bildgewinnung als auch Spektrographie von mehreren Ojekten vereint werden. Diese Masken werden in das für die Kühlung evakuierte Instrument durch eine spezielle Schleuse eingeführt. Im Inneren werden diese Masken in einer Vorratskassette, die insgesamt bis zu 33 Masken aufnehmen kann, aufbewahrt und wenn sie benötigt werden, durch einen Roboterarm in die richtige Position gebracht.

Dem Astronom stehen für seine Forschung außerdem verschiedenste Filter, die in zwei Filterrädern angebracht sind, zur Verfügung. Neben den genannten Optionen, die direkt von dem Benutzer des Instruments beeinflusst werden können, sind zusätzliche Mechanismen zur Kalibrierung im Inneren enthalten.

Als Detektor kommt ein von der Firma Rockwell entwickelter Detektor vom Typ *„HAWAII-2 HgCdTe“* zum Einsatz. Dieser Detektor verfügt über 2048x2048 Bildpunkten mit einer Größe von je 18x18 Mikrometern.

Insgesamt sind von der Steuersoftware mehr als 40 Motoren zu steuern. Da aufgrund der niedrigen Betriebstemperatur keine Positionsenkoder verwendet werden können, wird von der Steuersoftware verlangt, dass sie über andere Mechanismen die jeweiligen Elementpositionen errechnet.

In Abb. 1.3 ist der geplante Aufbau des Lucifer Instruments zu erkennen. Das mit (a) gekennzeichnete Element stellt die Schleuse für die Masken, die für die Spektrographie benötigt

werden, dar. Mit (b) ist die Elektronik, die zuständig für die Steuerung der Motoren ist, bezeichnet. Diese Elektronik stellt die Schnittstelle zwischen Instrument und Steuersoftware dar. Eines der beiden vorhandenen Filterräder ist mit dem Buchstaben (c) beschriftet. Die Struktur, die den einzelnen Elementen als Montagebasis dient, ist mit (d) in der Abb. 1.3 bezeichnet. Im Zentrum des Instruments befinden sich die mit (e) beschrifteten Wechseloptiken, die ebenfalls wie die Filter auf einem Rad montiert sind. Der Buchstabe (f) kennzeichnet die Fokalebene des Instruments. Unter der Fokalebene befindet sich der für den Einsatz der Masken zuständige Roboterarm, (g). Mit (h) ist der außen am Instrument angebrachte Kryostat bezeichnet. Die für die Auswahl der spektrographischen Auflösung sowie den normalen Bildgewinnungsprozess zuständige Einheit ist mit (i) beschriftet. Zuletzt sind noch die für die Kühlung nötigen Hitzeschilde zu nennen. Diese sind mit (j) bezeichnet.

Alle Informationen zu dem Lucifer Projekt sind den Internetseiten der LSW unter `http://www.lsw.uni-heidelberg.de/projects/Lucifer/` sowie [Mea00] entnommen. In Kapitel 2 wird das Lucifer Instrument näher beschrieben. Dort werden gezielt Themen, die für das virtuelle astronomische Instrument wichtig sind, behandelt.

Nachdem das Lucifer Projekt sowie das dazugehörige Großteleskop, beschrieben worden sind, werden als nächstes die Notwendigkeit und Aufgaben des virtuellen Instruments behandelt.

1.3 Notwendigkeit / Aufgaben eines virtuellen Instruments

Wie aus den vorangegangenen Unterkapiteln hervorgeht, ist das Lucifer Projekt ein im internationalen Rahmen eingebettetes und von mehreren deutschen Instituten durchgeführtes Vorhaben. Dabei liegt die Aufgabe des astronomischen Instituts der Universität Bochum darin, die Steuersoftware für das Lucifer Instrument zu entwickeln. Hierbei treten jedoch zwei Problematiken auf. Zum einen existiert eine räumlich Distanz zwischen dem astronomischen Institut Bochum und den anderen Instituten, die für den Instrumentenbau zuständig sind. Zum anderen ergibt sicht aus der Zeiteinteilung, dass nach Fertigstellung des Instruments nur eine sehr kurze Testphase für die Software zur Verfügung steht. Somit muss die Software möglichst gleichzeitig mit dem astronomischen Instrument fertiggestellt werden. Um dies realisieren zu können, ist es nötig eine Möglichkeit zu finden, die Software schon von Anfang an an einem virtuellen Instrument zu testen. Dieses Testen muss deswegen geschehen, da während jedes Softwareentwicklungsprozesses Fehler gemacht werden. Gerade bei der Entwicklung der Steuersoftware eines so komplexen Instruments muss vermehrt mit Fehlern gerechnet werden. Dabei entstehen laut [Per00] 64% der Fehler in der Analyse- und Designphase, und nur die restlichen 36% machen Programmierfehler aus. Nach einer Studie von IBM werden durch Testen während der Analyse- und Designphase ca. 50% der dort gemachten Fehler entdeckt. Ein Testen nach der Implementierung bringt insgesamt ca. 80% der in der Software enthaltenen Fehler zum Vorschein. Das Beseitigen dieser Fehler ist jedoch, verglichen mit Fehlern die während der Analyse- und Designphase entdeckt und behoben wurden, ca. 10fach so zeitaufwendig bzw. teuer. Für Fehler, die erst bei der späteren Benutzung entdeckt werden, gilt sogar, dass ca. 100mal mehr Zeit bzw. Geld zur Behebung aufgewendet werden müssen (vergleiche [Per00]).

Da durch Testen der Steuersoftware in der Analyse- und Designphase im Idealfall nur ca. 50% der Fehler aufspürt werden können und das Beheben von Fehlern der Steuersoftware, wenn sie sich bereits im Einsatzt befindet, sehr kostspielig[3] ist, muss die Testphase nach der Implementierung besonders gründlich ausfallen. Das dafür benötigte virtuelle Instrument sollte folgende Gesichtspunkte erfüllen:

[3]Damit sind Kosten für die Anreise sowie Kosten die durch den Ausfall des Instrumentes entstehen gemeint.

Schnittstellenkonformität: Die Schnittstellen zwischen Instrument und Software müssen weitestgehend erfüllt sein. Neben der Umsetzung der über die Schnittstelle gehenden Befehle ist es wichtig, dafür zu sorgen, dass entsprechende Antworten generiert werden. Bei der Erzeugung von Antworten ist besonders auf Einhaltung der richtigen Antwortzeitpunkte zu achten.

Zeitäquivalenz: Das zeitliche Verhalten der einzelnen Elemente des Instruments muss nachgebildet werden, so dass zeitabhängige Antworten, die über die Schnittstelle mit der Software ausgetauscht werden, zum richtigen Zeitpunkt gesendet werden. Hierbei sind insbesonders die Fahrtzeiten der Motoren zu nennen, die durch Fehlerfaktoren beeinflusst werden.

Fehlerfaktoren: Fehlerfaktoren, die auch beim echten Instrument existieren, müssen durch Zufallsfunktionen nachempfunden werden. So können z.B. zu hohe Geschwindigkeiten oder Drehmomente zu einem Fehlverhalten der Elemente im Instrumenteninneren führen, das entsprechend dargestellt werden soll.

Visualisierung: Eine Visualisierung der komplexen Elementinteraktion ist notwendig, da aufgrund der hohen Komplexität der im inneren interagierenden Elemente ein Verständnis des Instrumentenzustandes nur anhand von reinen Positionsdaten schwer möglich ist. Hinzu kommt, dass bei dem realen Instrument aufgrund seines geschlossenen Aufbaus solche Möglichkeiten nicht gegeben sind, jedoch für die Softwareentwicklung sehr wichtig sind.

Flexibilität: Eine große Flexibilität muss gegeben sein, damit Änderungen im Instrumentendesign auch in das virtuelle Instrument übernommen werden können. Außerdem können so auch Instrumente ähnlichen Designs nachgebildet werden.

Zusammenfassend kann man sagen: „Das virtuelle Instrument soll das tatsächliche Instrument so gut nachbilden, dass über die Schnittstelle das virtuelle von dem realen Instrument möglichst nicht zu unterscheiden ist.“ In dem folgenden Kapitel werden technische Grundlagen behandelt, die notwendig sind, um die Funktionsweise des Lucifer Instruments zu verstehen.

Und sie bewegt sich doch.

Galileo Galilei

Kapitel 2

Technischer Hintergrund

Dieses Kapitel ist dazu gedacht, den technischen Hintergrund der Diplomarbeit zu erläutern. Dazu gehören zum einen nähere Informationen zu dem Lucifer Instrument, zum anderen allgemeinere technische Beschreibungen. Diese sind dazu notwendig zu verstehen, welche Elemente sich wie und auf welche Weise bewegen lassen. Als erstes werden für die spätere Umsetzung des virtuellen astronomischen Instruments notwendige Informationen zu Lucifer gegeben.

2.1 Lucifer

Neben der in Kapitel 1 gegebenen Übersicht zu dem Lucifer Projekt sind weitere Informationen für diese Arbeit wichtig. Wie aus Kapitel 1 ersichtlich ist, setzt sich das Lucifer Instrument aus einer Reihe bewegbarer Elemente zusammen. Dazu sind ca. 40 Elektromotoren gezielt anzusteuern. Diese Ansteuerung geschieht durch eine vom Max Planck Institut für Astronomie entwickelte Elektronik. Um die Bewegungen der Elemente im Instrumenteninneren nachvollziehen zu können, müssen die Ansteuervorgänge der Elektronik nachgebildet werden. Als nächstes wird deswegen die für das Lucifer Projekt verwendete Elektronik vorgestellt.

2.1.1 Elektronik

Die Ansteuerung der Motoren des Lucifer Instrumentes erfolgt über die in Abb. 2.1 gezeigte Steuerelektronik. Diese Elektronik wird später außen am Instrument befestigt (siehe (b) in Abb. 1.3). Dabei setzt sich die Steuerelektronik, wie Abb. 2.1 zu entnehmen ist, aus mehreren Elektronikmodulen zusammen. Das mit SMC8-NANO (*stepping motor controler for 8 steppingmotors*) gekennzeichnete Modul ist zuständig für die Erzeugung der benötigten Steuersignale. Links daneben befindet sich ein Modul, in dem die zur Steuerung der Motoren benötigten Ströme entsprechend verstärkt werden. Zusätzlich sind in diesem Modul die Anschlüsse für die in Unterkapitel 2.2.2 beschriebenen Referenzschalter enthalten. Insgesamt können an diesem Modul 4 Motoren angeschlossen werden. Die daneben befindlichen Module werden für den Anschluss von zusätzlichen Referenzschaltern benötigt. Auf der rechten Seite befinden sich Module, die die benötigte Spannungsversorgung bereitstellen. Alle Module werden in einem nach Industriestandard gefertigtem 19“ Gehäuse untergebracht.

Da die in Abb. 2.1 dargestellte Steuerelektronik nur 4 Motoren ansteuern kann, wird für die spätere Ansteuerung der ca. 40 Motoren eine wesentlich umfangreichere Elektronik benötigt. Die Kommunikation zwischen der Steuerelektronik und der Software findet über die in Abb. 2.1 gezeigte serielle RS232 Schnittstelle statt. Damit die Anzahl der seriellen Schnittstellen klein gehalten werden kann, können die SMC8-NANO Elektronikmodule über ein Bussystem miteinander verbunden werden. Die serielle Anbindung der Elektronik an den Computer, der die Steuerungsoftware enthält, wird im nächsten Unterkapitel beschrieben.

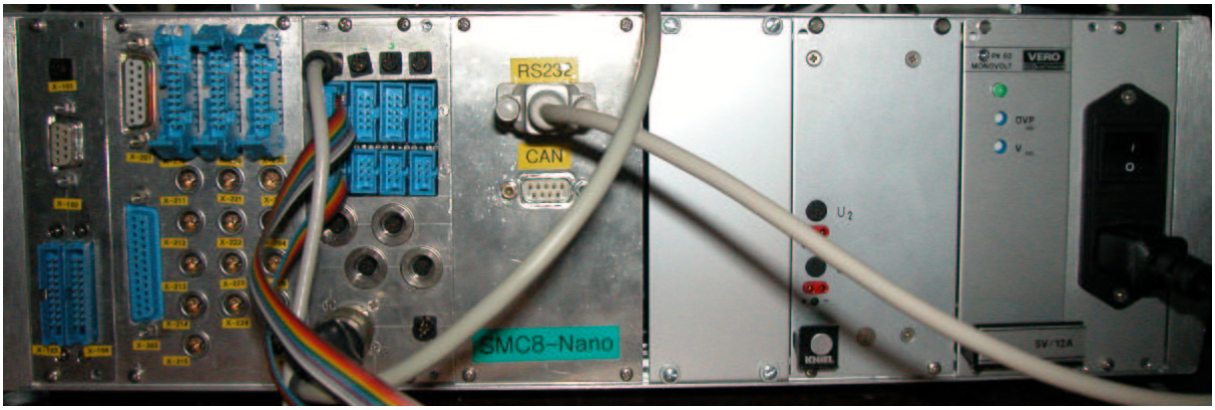


Abbildung 2.1: Steuerelektronik

2.1.2 Computerschnittstelle

Die Steuerelektronik empfängt über die serielle Schnittstelle die zur Steuerung notwendigen Kommandos. Diese Befehle können das Verhalten der Steuerelektronik konfigurieren, Fahrbefehle ausführen lassen oder Anfragen an die Elektronik sein (siehe Anhang C). Die Kommunikation des Steuercomputers mit der Elektronik kann über eine einfache serielle Verbindung stattfinden. Da zwischen dem Lucifer Instrument und dem entsprechenden Computer eine galvanische Trennung[1] gefordert ist und außerdem zwischen dem Teleskop und dem Kontrollraum eine große Strecke überbrückt werden muss, wurde eine andere Verbindungsart gewählt. Anstelle einer normalen seriellen Kabelverbindung wird eine Glasfaserverbindung verwendet. Damit die Kommunikation mit dem Computer auf sehr einfache Art geschehen kann, werden die seriellen Schnittstellen der Steuerelektronik mit einem Portserver verbunden. Dieser setzt die seriellen Verbindungen auf eine Netzwerkverbindung um. Dazu wird das gängige TCP/IP Protokoll verwendet, wobei jedem seriellen Anschluss eine entsprechende Portadresse zugewiesen wird (vergleiche [Com01]). Die Anbindung des Portservers an den Rechner geschieht über die bereits genannte Glasfaserstrecke. Somit findet die gesamte Kommunikation mit dem Instrument über ein Rechnernetz statt. Für die in Kapitel 1 genannte Schnittstellenkonformität wurde diese Netzwerkschnittstelle gewählt. Somit ist die spätere Verwendung des virtuellen astronomischen Instruments auch ohne den Einsatz eines Schnittstellenkonverters möglich.

Als nächstes werden technische Grundlagen der für das Instrument wichtigen Mechanik beschrieben.

2.2 Mechanik

In diesem Unterkapitel werden Grundlagen bezüglich der Mechanik des Lucifer Instruments besprochen. Auf diese Informationen bezieht sich ein Teil der theoretischen Überlegungen in Kapitel 3. Neben den verwendeten Elektromotoren werden Referenzschalter, Rastmechanismen sowie unterschiedliche Bewegungsarten von mechanischen Elementen beschrieben. Dabei wird neben der Technik auch die Anwendung der vorgestellten Instrumentenbestandteile betrachtet. Als erstes werden die verwendeten Elektromotoren behandelt.

2.2.1 Schrittmotoren

In Lucifer werden Schrittmotoren zur Bewegung der Elemente genutzt. Schrittmotoren unterscheiden sich in ihrer Funktionsweise von herkömmlichen Elektromotoren. Ein Elektromotor

[1] Es darf keine Verbindung aus stromleitenden Elementen existieren.

bewegt sich dadurch, dass ein durch Strom hervorgerufenes Magnetfeld in Wechselwirkung mit einem Dauermagneten tritt. Durch sinusidiale Oszillation des Stroms wird eine Rotationsbewegung erzeugt, die in dem abwechselnden Anziehen und Abstoßen der stromführenden Spulen von den Dauermagneten begründet liegt.

Bei einem Schrittmotor sind die Dauermagneten und Spulen so angeordnet, dass durch ein geschicktes Ansteuern der Spulen nur einzelne Schritte ausgeführt werden. In Abb. 2.2 ist an einem schematischen Schrittmotor eine solche Ansteuerfolge gezeigt. Unter den Schrittmoto-

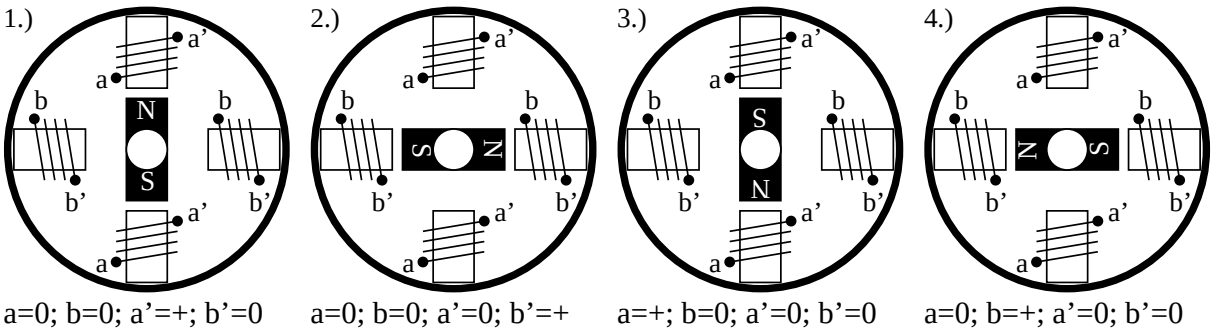


Abbildung 2.2: Funktionsweise eines Schrittmotors

ren der jeweiligen Phase ist die entsprechende Ansteuerung abzulesen. Dabei bezeichnet ein „+“, dass eine positive Spannung an der jeweiligen Spule anliegt. Dies nennt sich unipolare Beschaltung. Um ein höheres Drehmoment zu erhalten, kann der Schrittmotor bipolar angesteuert werden. Das heißt, dass neben der positiven Spannung an einer Spule eine negative Spannung an der gegenüberliegenden angelegt wird.

Der in Abb. 2.2 gezeigte Schrittmotor hat eine Schrittgröße von 90°. Durch den Aufbau eines Schrittmotors kann dessen Ausrichtung im Gegensatz zu einem normalen Elektromotor sehr genau bestimmt werden. Die Genauigkeit eines Schrittmotors wird durch seine Schrittgröße definiert. Die Schrittmotoren die bei Lucifer Verwendung finden haben eine Auflösung von 1,8° bzw. 0,9°. Das entspricht einer Anzahl von 200 bzw. 400 Schritten pro Motorumdrehung. Abhängig von der Frequenz, mit der die Schritte ausgeführt werden, wird aus einer ruckartigen Motorbewegung eine fließende. Um dies zu unterstützen beherrscht die Steuerelektronik das Mikroschrittverfahren. Dabei werden die sehr großen Einzelschritte in Mikroschritte unterteilt. Durch gezielte Ansteuerung zweier benachbarter Spulen, wobei die Spannung der einen Spule proportional zu der Spannungszunahme der Anderen abnimmt, wird der Schrittmotor langsam geführt, ohne ruckartig die Ausrichtung zu ändern. Wenn man sich die Ansteuerung mit „a=0; b=0; a'=+; b'=+“ als Zwischenschritt zwischen Phase 1 und Phase 2 in Abb. 2.2 vorstellt, wird ersichtlich, dass dies zu einer Halbierung des sonst 90° betragenen Schrittes führen würde.

Eine feste Position kann der Schrittmotor jedoch nur an den durch den Aufbau vorgegebenen Positionen annehmen. Das Halten einer Position zwischen zwei Spulen würde aktive Regelkreisläufe erfordern und ist somit für Lucifer nicht möglich. Werden die Spulen eines Schrittmotor stromlos geschaltet, kann sich dieser frei von außen bewegen lassen.

Neben der Erzeugung der richtigen Ansteuerung der Schrittmotoranschlüsse und der Ermöglichung einer weichen Bewegung im Mikroschrittbetrieb ist die Steuerelektronik außerdem für die Berechnung von Fahrprofilen zuständig. Diese Fahrprofile sind dazu notwendig, damit das Drehmoment, das während des Anfahrens auf den Motor einwirkt, ausgeglichen wird. Außerdem soll ein Nachlaufen nach der Beendigung einer Bewegung - bewirkt durch die Massenträgheit des von dem Motor angetriebenen Elements - verhindert werden. Sowohl ein zu hohes Anfangsdrehmoment als auch ein Nachlaufen können zu Positionierungsfehlern eines Schrittmotors führen. Die in Abb. 2.3 gezeigten Fahrprofile werden von der bereits beschriebenen Steuerelektronik unterstützt. Das erste Fahrprofil entspricht einer Fahrt ohne Beschleunigungs-

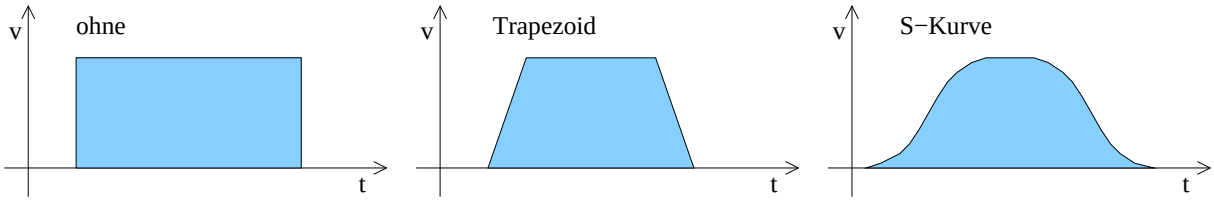


Abbildung 2.3: Verschiedene Fahrprofile

und Abbremsphase mit gegebener Geschwindigkeit in Schritten pro Sekunde. Zur Vereinfachung wurden die einzelnen Schritte zusammengefasst anstatt sie einzeln darzustellen. Die zweite Geschwindigkeitskurve stellt eine Bewegung mit linearer Beschleunigung und Abbremsung dar. Dieses Fahrprofil wird aufgrund seiner Form Trapezoid genannt und minimiert das beschriebene Risiko, Positionierungsfehler zu erzeugen. Die für eine Bewegung optimale Geschwindigkeitskurve stellt die so genannte S-Kurve dar. Sie verfügt über eine sich steigernde Beschleunigungsphase, die, sobald sie eine gewünschte Beschleunigung errreicht hat, diese beibehält. Vor Erreichen der Plateau- bzw. Arbeitsphase wird die Beschleunigung reduziert, um ein Überschwingen zu verhindern. Die Abbremsphase gestaltet sich analog zu der Beschleunigungsphase. Durch das Verwenden der S-Kurve wird eine sehr gute Positionierungsgenauigkeit erreicht, da das Risiko des Nachlaufens sowie des Auslassens von Schritten minimiert wird. Nähere Informationen zu Schrittmotoren und ihrer Anwendung finden sich unter [Kot74].

Für das Erstellen der richtigen Fahrprofile benötigt die Steuerelektronik entsprechende Parameter, die auch für das virtuelle astronomische Instrument wichtig sind, damit dieses die Fahrprofile nachbilden kann. Neben einem Parameter, der eines der drei in Abb. 2.3 gezeigten Profile auswählt, werden die Startgeschwindigkeit, die gewünschte Geschwindigkeit, die Startbeschleunigung, die gewünschte Beschleunigung sowie die Beschleunigungsänderung benötigt. Dabei haben die Parameter folgende Bedeutung:

- *Startgeschwindigkeit* v_0
  Dies ist der Geschwindigkeitsoffset, der angibt, mit welcher Geschwindigkeit ein Fahrprofil beginnt. Wenn ohne Beschleunigungs- und Abbremsphase gefahren wird, ist dieser Parameter ohne Bedeutung. In Abb. 2.3 sind alle Geschwindigkeitskurven mit einem Offset von 0 angegeben. Der Offset bewirkt eine Verschiebung entlang der mit v gekennzeichneten Geschwindigkeitsachse.

- *gewünschte Geschwindigkeit* v_{des}
  Die gewünschte Geschwindigkeit versucht das Fahrprofil in seiner Plateauphase zu erreichen. Diese kann aufgrund der Wahl der anderen Parameter manchmal nicht erreicht werden, da während der notwendigen Beschleunigungsphase die zu fahrende Strecke überschritten würde.

- *Startbeschleunigung* a
  Die Startbeschleunigung bestimmt für sowohl das Trapezoid als auch das S-Kurvenprofil die Beschleunigung zu Beginn der Bewegung.

- *gewünschte Beschleunigung* a_{des}
  Während der Beschleunigungsphase des S-Kurvenprofils wird versucht, die gewünschte Beschleunigung zu erreichen. Nach Erreichen dieses Werts folgt eine lineare Beschleunigungsphase, bevor die Beschleunigung vor Erreichen der gewünschten Geschwindigkeit auf ihren Ausgangswert reduziert wird. Wie auch bei der gewünschten Geschwindigkeit ist es möglich, dass die gewünschte Beschleunigung nicht erreicht wird.

- *Beschleunigungsänderung* $\dot{a}$
  Dieser Parameter bestimmt die Änderungsrate, mit der bei Verwendung des S-Kurvenprofils die Beschleunigung modifiziert wird.

In Anhang C finden sich die Befehle, mit denen die Fahrprofile der Steuerelektronik selektiert und konfiguriert werden können. Zusätzlich benötigt die Steuerelektronik Parameter über die Schrittgröße, die Mikroschrittunterteilung sowie das Übertragungsverhältnis zwischen Schrittmotor und angetriebenem Element, um die entsprechenden Fahrbefehle durch die Schrittmotoren ausführen zu lassen.

Neben den für die Bewegung innerhalb des Instruments zuständigen Schrittmotoren sind die Referenzschalter elementare Bestandteile von Lucifer, da diese die einzige Möglichkeit sind, Informationen aus dem Instrumenteninneren zu bekommen.

2.2.2 Referenzschalter

Um Informationen aus dem Inneren des Instruments zu bekommen, werden kleine Referenzschalter angebracht. Diese werden in zwei Klassen unterteilt. Die eine Klasse von Schaltern wird durch die bewegungslimitierenden Endschalter gegeben. Die Aufgabe der Endschalter liegt darin, eine obere und eine untere Bewegungsgrenze zu definieren. So darf z.B. der in Kapitel 1 erwähnte Maskenmechanismus nicht beliebig lange in eine Richtung verschoben werden, da zum einen Lucifer endlich ist, zum anderen die Kollisionsgefahr mit anderen Elementen besteht. Manche rotierenden Elemente des Lucifer Instruments verfügen über Endschalter, da bei ihnen ein endloses einseitiges Drehen z.B. durch Verbindungskabel nicht möglich ist.

Neben den Endschaltern, deren Aufgabe darin besteht, für Sicherheit im Instrumenteninneren zu sorgen, verfügt Lucifer über eine zweite Klasse. In dieser Klasse sind Schalter enthalten, die Positionskalibrierungen an dem Element, an dem sie befestigt sind, ermöglichen. Diese Schalter geben eine Nullposition für das jeweilige Element an. Durch eine von der Steuerelektronik implementierte Kalibrierungsmethode kann das jeweilige Element mit seinem Referenzschalter synchronisiert werden. Dazu fährt das Element in einer durch zusätzliche Parameter gegebenen Geschwindigkeit und Richtung los. Das Element wird solange bewegt, bis es entweder die Nullposition findet oder einen Endschalter erreicht. Wird ein Endschalter erreicht, wird die Richtung geändert und somit in der entgegengesetzten Richtung gesucht.

Da mit Hilfe der Kalibrierung, den Endschaltern und der richtigen Ansteuerung der Schrittmotoren keine 100% genaue Positionierung gewährleistet werden kann, wird ein Rastmechanismus verwendet.

2.2.3 Rasten

Da für manche Elemente sehr hohe Positioniergenauigkeit gefordert wird, werden Rasten auf diese Elemente montiert. Eine Raste stellt dabei eine Einkerbung mit zwei Flanken dar. Zum Positionieren wird ein Rad, welches federnd gelagert ist, benutzt. Der in Abb. 2.4 gezeigte Einrastvorgang findet auf einer ebenen Fläche statt. Die Funktionsweise einer Raste kann einfach auf anders geformte Oberflächen übertragen werden, deswegen wird dieser Vorgang nur anhand der übersichtlicheren ebenen Fläche erläutert. In Phase 1 von Abb. 2.4 erkennt man, wie sich das zu justierende Element durch einen nicht dargestellten Motor bewegt. Dabei drückt eine Feder ein rollend gelagertes Rad auf die Oberfläche dieses Elements. Nachdem die Bewegung durch den Motor beendet ist, befindet sich das Element in Phase 2. Hierbei drückt die Feder das Rad auf die vordere Flanke der Raste. Wenn die Federkraft ausreichend ist, resultiert aus der Neigung der Flanke eine Bewegung des Rastenmittelpunktes zur Position des Rades. Nach Abschluss dieses Einrastvorganges befindet sich das Element in Phase 3. Die Raste und das

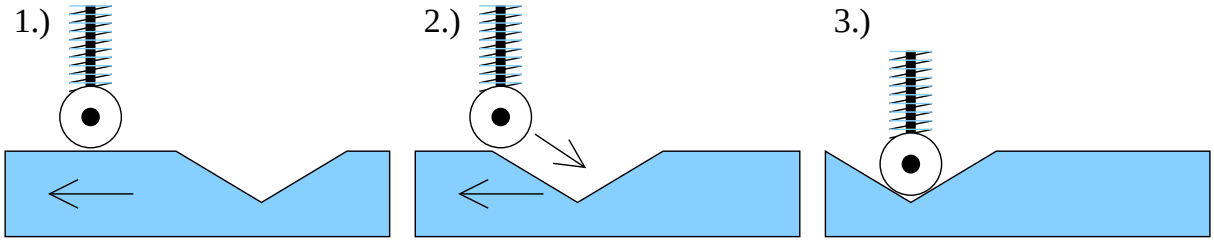


Abbildung 2.4: Rastmechanismus

Rad sind in einem stabilen Zustand aufeinander synchronisiert. Durch gezielte Positionierung des Rades in Bezug zu dem zu justierenden Element sowie das Vorhandensein von Rasten auf einem Element, kann somit eine sehr genaue Positionierung erreicht werden.

Im nächsten Kapitel folgen theoretische Überlegungen zur Umsetzung der beschriebenen Technik in ein Simulationsmodell sowie der Nachbildung der Kommunikationsschnittstelle.

Theorien sind gewöhnlich Übereilungen eines ungeduldigen Verstandes, der die Phänomene gern los sein möchte und an ihrer Stelle deswegen Bilder, Begriffe, ja oft nur Worte einschiebt.

Johann Wolfgang von Goethe

Kapitel 3

Theoretische Vorüberlegungen

Nachdem ein kleiner Überblick über wichtige technische Abläufe innerhalb des Lucifer Instruments gegeben wurde, sind in diesem Kapitel theoretische Vorüberlegungen zur Realisierung eines virtuellen astronomischen Instruments enthalten. Im ersten Unterkapitel wird der gewählte Ansatz zur Nachbildung der realen Bewegungen der Mechanik erklärt. Danach wird sich mit der Emulation der Elektronik, einer Nachbildung der Firmware, befasst. Anschließend daran werden die theoretischen Vorüberlegungen zu einer entsprechenden Visualisierung behandelt. Diese Dreiteilung in Simulation, Emulation und Visualisierung findet man auch im nächsten Kapitel über die Umsetzung, so dass ein Bezug zu den theoretischen Vorüberlegungen einfacher ermöglicht wird.

3.1 Simulation der Mechanik

Bevor man sich der Simulation einer Sache zuwendet, ist es wichtig, den Begriff des Simulators zu verstehen. [Ins82, Simulator, der: Gerät, in dem künstlich die Bedingungen u. Verhältnisse herstellbar sind, wie sie in Wirklichkeit bestehen. ...] Wenn man demnach etwas simulieren möchte, muss man die reale Welt bestmöglich nachbilden. Diese mathematische Modellbildung ist Bestandteil der Informatikvorlesung zum Thema Simulation. Anstelle einer Beschreibung der Modellbildungstechniken sowie der unterschiedlichen Simulationsmethoden zu geben, die über den Umfang dieser Arbeit hinausgehen würden, wird auf Literatur wie z.B. [CL99, LK99, Rub98] verwiesen.

Im Rahmen der Modellierung des zu simulierenden astronomischen Instruments ist es notwendig, die zu simulierenden Parameter der Mechanik zu bestimmen.

3.1.1 Definition der zu simulierenden Instrumentenparameter

Bei jeder Simulation ist es wichtig, die zu simulierenden Parameter einzuschränken, um das Simulationsmodell berechenbar zu halten. Würden bei dem Simulationsmodell alle physikalischen Parameter, wie z.B. das Verhalten von Materialien bei extrem tiefen Temperaturen, berücksichtigt, wäre es unmöglich ein Modell zu formulieren. Die genaue Nachbildung der Realität ist in einer Simulation nicht möglich, da immer gewisse Bereiche noch genauer spezifiziert werden können. Würde man das virtuelle astronomische Instrument bis auf Quantenebene nachbilden, ständen außerdem die dafür notwendigen Rechnerkapazitäten nicht zur Verfügung.

Um das virtuelle astronomische Instrument nachbilden zu können, beschränkt sich die Simulation der Mechanik auf einige Parameter. Die Ausrichtung des Schrittmotors durch den ein Element angetrieben wird, ist als einer der zu simulierenden Werte zu betrachten. Die daraus resultierende Elementenbewegung ist nicht für die Simulation des astronomischen Instruments von Bedeutung. Elementkollisionen im Instrumenteninneren können aufgrund des

Designs nicht vorkommen. Somit wird die Elementausrichtung nur von den Visualisierungskomponenten benötigt und nicht in der Instrumentensimulation betrachtet. Bei der Ausrichtung des Schrittmotors müssen grundsätzlich zwei Werte unterschieden werden. Zum einen die tatsächliche Stellung des Schrittmotors, zum anderen die Stellung, von der die Steuerelektronik ausgeht. Da aufgrund der kryogenischen Kühlung auf 80°K ein Positionsenkoder nicht verwendet werden kann, bezieht die Steuerelektronik ihre Positionsinformationen aus der Anzahl erzeugter Schrittsignale. Diese erkennt weder, wieviel Schritte der Schrittmotor tatsächlich ausgeführt hat, noch wieviel zusätzliche Schritte durch eine externe Elementbewegung, wie z.B. einen Einrastvorgang, erzeugt worden sind. Sowohl die reale Motorposition als auch die von der Steuerelektronik erwartete Position sind für das virtuelle astronomische Instrument wichtig.

Einen weiteren wichtigen Parameter für die Modellbildung stellt das Übersetzungsverhältnis zwischen Motor und angetriebenem Element dar. Obwohl die Elementbewegung selbst nicht simuliert wird, benötigt man dieses Verhältnis, um die für die Simulation wichtigen auf dem Element montierten Referenzschalter und Rasten entsprechend nachzubilden.

Die Referenzschalter sind in Form der Endschalter ein Limit für die reale Motorbewegung. Diese Aufgabe müssen sie auch in der Simulation erfüllen. Dort stellen sie ein Maximum bzw. Minimum für die simulierte Motorausrichtung dar. Außerdem muss in der Simulation berücksichtigt werden, dass bei einem Erreichen dieser Endschalter eine entsprechende Nachricht erzeugt wird. Der Kalibrierungschalter muss ebenfalls in der Simulation nachgebildet werden. Nur durch Festlegen seiner Position wird der virtuellen Steuerelektronik ermöglicht, die Kalibrierungsbewegung durchzuführen und so den ebenfalls simulierten erwarteten Motorwinkel mit dem Reellen zu synchronisieren. Dabei definiert die Breite des Schalters ein Intervall, in dem die Steuerelektronik den Motor an der richtigen Kalibrierungsposition erkennt.

Mit den Rasten und den durch sie hervorgerufenen Einrastvorgängen ist ein weiterer Parameter für die Simulation zu nennen. Durch den Einrastvorgang wird nur die tatsächliche Motorausrichtung innerhalb des Simulationsmodells beeinflusst. Die erwartete Motorausrichtung bleibt in der Simulation unverändert, da von der virtuellen Steuerelektronik keine Schrittsignale erzeugt wurden. Bei den zu simulierenden Rastvorgängen ist darauf zu achten, dass nur nach Stromlosschaltung eines Schrittmotors dieser von außen bewegt werden kann[1].

Bevor auf die Modellierung des virtuellen astronomischen Instruments eingegangen wird, müssen die Anforderungen an die Simulation betrachtet werden.

3.1.2 Anforderungen an die Simulation der Mechanik

Aufgrund der in Kapitel 1 genannten Aufgaben des virtuellen astronomischen Instruments ergeben sich die Anforderungen, die an die Simulation der Mechanik gestellt werden. Fast alle Simulationen werden dazu benutzt, zeitliche Abfolgen nachzubilden. Eine der genannten Aufgaben ist es, dafür zu sorgen, dass die Ausgaben sowohl des echten als auch des virtuellen Instruments synchron erfolgen. Diese Zeitäquivalenz der Ausgaben muss nur im Bereich einer zehntel Sekunde liegen, da durch die Datenkommunikation über ein Rechnernetz ebenfalls Verzögerungen, wie z.B. Vermittlungszeiten, auftreten. Neben der Simulation der Instrumentenparameter ist somit die Zeit von besonderer Bedeutung. Ein rein ereignisorientierter Simulationsansatz ist somit nicht zu verwenden, da die dort verwendete Modellzeit nicht in Verbindung mit der Realzeit steht.

Als zweite Anforderung, muß die Simulation ein Fehlverhalten des Motors nachbilden können. Da das tatsächliche Fehlverhalten eines Schrittmotors nicht zu modellieren ist, wird versucht, durch Zufallszahlen, die einer vorgegebenen Verteilung entsprechen, das Fehlverhalten zu approximieren. Ab einer vorgegebenen Geschwindigkeit soll die Simulation der realen Mo-

[1] Ausgenommen ist hier das Drehen unter extremer Krafteinwirkung.

torgeschwindigkeit durch diese Zufallsfunktion moduliert werden. Dabei ist es nicht wichtig, reproduzierbare Pseudozufallszahlen, wie in [Knu98a] beschrieben, zu verwenden, sondern es ist ausreichend, die in Abhängigkeit von der internen Uhr erzeugten Zufallszahlen zu nutzen. Außerdem ist dafür zu sorgen, dass die simulierte reelle Motorgeschwindigkeit bei Überschreiten einer gewissen Beschleunigung und des damit verbunden Drehmomentes auf Null abfällt.

Nachdem die für die Simulation wichtigen Parameter eingegrenzt worden sind und die Anforderungen für die Simulation der Mechanik festgelegt wurden, wird im nächsten Unterkapitel der gewählte Simulationsansatz behandelt.

3.1.3 Gewählter Simulationsansatz

Als Simulationsansatz wurde die ereignisorientierte Methode gewählt und entsprechend modifiziert. Die zu simulierenden Ereignisse stellen die Motorbewegung dar. Wie bereits in Unterkapitel 3.1.1 beschrieben wurde, findet eine Beschränkung auf die Simulation der Motorbewegung statt. Diese Motorbewegungen können, aufgrund der Unterscheidung zwischen realem Winkel und dem von der Steuerelektronik Erwarteten in drei Klassen eingeteilt werden:

1. Bewegungen, die sowohl den realen als auch den erwarteten Motorwinkel betreffen. Diese Bewegungen werden von der Steuerelektronik erzeugt und verändern - vorausgesetzt die maximal zulässige Beschleunigung wird nicht überschritten - sowohl den internen Schrittzähler der Elektronik als auch die tatsächliche Motorposition. Dabei ist zu beachten, dass, durch eine Zufallsfunktion gesteuert, die tatsächliche Motorstellung von der Erwarteten abweichen kann.

2. Bewegungen, die nur den erwarteten Motorwinkel betreffen. Diese Art von Ereignis tritt ein, wenn sich der reale Motorwinkel aufgrund eines zu hohen Drehmomentes - hervorgerufen durch eine zu starke Beschleunigung - nicht verändert.

3. Bewegungen, die nur den realen Motorwinkel betreffen. Werden Bewegungen von außen auf den Schrittmotor ausgeübt, wie es z.B. durch den Einrastvorgang geschehen kann, so verändert sich nur der reale Motorwinkel. Da diese Bewegung nicht von der Steuerelektronik hervorgerufen wurde, wird diese Veränderung von der Elektronik nicht berücksichtigt. Somit bleibt der erwartete Motorwinkel unverändert.

Die Motorbewegung lässt sich durch die folgende Formel, die eine beschleunigte Bewegung beschreibt, darstellen.

$$v(t) = v_0 + at + \frac{\dot{a}t^2}{2}, \tag{3.1}$$

Dabei bezieht sich die Formel 3.1 auf die Geschwindigkeit des Schrittmotors in Abhängigkeit von einer Startgeschwindigkeit[2] v_0, einer Beschleunigung a und einer Beschleunigungsänderung $\dot{a}$. Um die während eines Zeitintervalls $[t_0, t]$ erfolgte Winkeländerung zu erhalten, muss die Fläche zwischen der zuvor beschriebenen Formel und der Zeitachse für das vorgegebene Intervall berechnet werden. Das sich daraus ergebende Integral kann, mit $t_0 = 0$, durch folgende Formel gelöst werden:

$$\omega(t) = v_0 t + \frac{at^2}{2} + \frac{\dot{a}t^3}{6} + \omega_0 \tag{3.2}$$

Der Winkel ω, um den sich der Motor dreht, lässt sich, unter Vernachlässigung von ω_0, somit leicht berechnen. Um den drei genannten Bewegungsklassen gerecht zu werden, müssen die Winkel jeweils für die reelle und die erwartete Motorstellung getrennt berechnet werden.

[2] Alle verwendeten Symbole werden in Anhang B nochmals beschrieben.

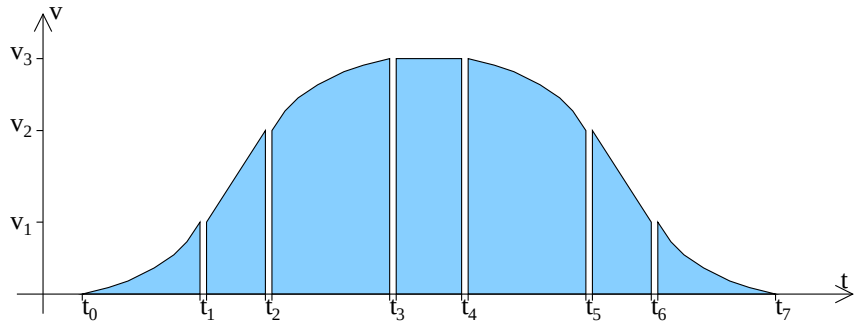


Abbildung 3.1: Zerlegtes Fahrprofil

Gegebenenfalls muss die durch die Zufallsfunktion hervorgerufene Geschwindigkeitsmodifikation berücksichtigt werden. Dazu wird mittels der Geschwindigkeitsformel 3.1 das Zeitintervall berechnet, in dem die zulässige Motorgeschwindigkeit überschritten wird und die in diesem Zeitintervall verminderte Winkeländerung subtrahiert. Für den Fall, dass die maximal zulässige Beschleunigung a_{sim} erreicht wird, muss ab Erreichen dieser die tatsächliche Motorbewegung angehalten werden. Somit ist für die reelle Motorbewegung nur der Zeitpunkt t_1, an dem die Formel

$$a(t) = a + \dot{a}t \tag{3.3}$$

den Wert von a_{sim} annimmt, zu berechnen und in die entsprechende Winkelformel 3.2 einzusetzen.

Da sich die von der Steuerelektronik erzeugten Fahrprofile nicht in einer geschlossenen Form durch die Geschwindigkeitsformel 3.1 ausdrücken lassen, muss ein Fahrprofil durch Aufteilung in darstellbare Intervalle beschrieben werden. Abb. 3.1 zeigt eine solche Zerteilung bezüglich des S-Kurvenprofils. Die mathematische Beschreibung hierzu liefert folgende Formel:

$$v(t) = \begin{cases} v_0 + a(t-t_0) + \frac{\dot{a}(t-t_0)^2}{2} & , t_0 \le t \le t_1 \\ v_0 + a(t_1-t_0) + \frac{\dot{a}(t_1-t_0)^2}{2} + a_{des}(t-t_1) & , t_1 \le t \le t_2 \\ v_0 + a(t_1-t_0) + \frac{\dot{a}(t_1-t_0)^2}{2} + a_{des}(t_3-t_1) - \frac{\dot{a}(t-t_2)^2}{2} & , t_2 \le t \le t_3 \\ v_{des} & , t_3 \le t \le t_4 \\ v_0 + a(t_1-t_0) + \frac{\dot{a}(t_1-t_0)^2}{2} + a_{des}(t_2-t_1) + a_{des}(t_5-t) - \frac{\dot{a}(t_5-t)^2}{2} & , t_4 \le t \le t_5 \\ v_0 + a(t_1-t_0) + \frac{\dot{a}(t_1-t_0)^2}{2} + a_{des}(t_6-t) & , t_5 \le t \le t_6 \\ v_0 + a(t_7-t) + \frac{\dot{a}(t_7-t)^2}{2} & , t_6 \le t \le t_7 \end{cases} \tag{3.4}$$

Mit Formel 3.4 lassen sich die Geschwindigkeiten in jedem Bereich einer S-Kurve sowie in allen dadurch ausdrückbaren Profilen, berechnen. Hierbei bezeichnet das neu hinzugekommende Symbol v_{des} die gewünschte Geschwindigkeit. Da sich Formel 3.4 aus Formel 3.1 ergibt, kann somit die Simulation der Motorenwinkel auf diese aufgebaut werden.

Wie bereits erwähnt, wird ein modifizierter ereignisorientierter Simulationsansatz verfolgt. Dazu werden die Ereignisse, die jeweils durch eine Bewegung nach Formel 3.1 repräsentiert werden, in einer zeitlich sortierten Liste verwaltet. Da im Gegensatz zu der ursprünglichen ereignis- bzw. prozessorientierten Simulation nur aufgrund der Realzeit Ereignisse generiert werden, kann es nicht vorkommen, dass Ereignisse zeitlich vor bereits in der Liste eingetragenen Ereignissen stattfinden. Somit ist eine einfache „*first in / first out*“ Liste ausreichend für die Ereignisverwaltung.

Werden neue Ereignisse in die Liste eingetragen, muss zuerst der aktuelle Motorwinkel berechnet werden (siehe Phase 1 Abb. 3.2). Dies geschieht dadurch, dass alle Ereignisse vom Anfang der Liste bis zu dem aktuellen Zeitpunkt berechnet werden. Bei dieser Berechnung wird zwischen dem reellen und dem erwarteten Winkel unterschieden. Gegebenenfalls wird das dabei

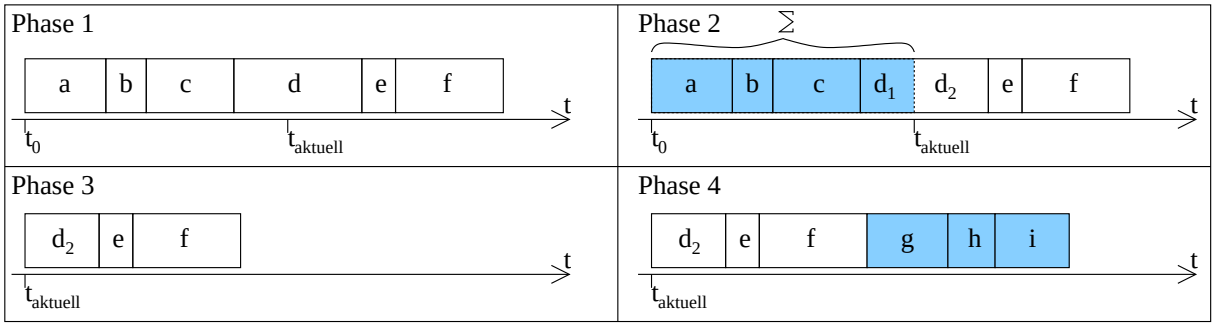


Abbildung 3.2: Simulationsschema

letzte Ereignis nur teilweise berechnet und um den betrachteten Bereich gekürzt (vergleiche Phase 2 Abb. 3.2).

Danach werden die berechneten Ereignisse aus der Liste gelöscht, da diese nicht mehr benögigt werden (siehe Phase 3 Abb. 3.2). Nach Bestimmung der aktuellen Ausrichtung des Motors und dem Löschen der berechneten Ereignisse, werden die neuen Ereignisse eingefügt (siehe Phase 4 Abb. 3.2). Dabei ist darauf zu achten, dass, wenn diese Ereignisse von der Steuerelektronik generiert worden sind, eventuell in der Liste vorrangehende und durch externe Motorbewegungen erzeugte Ereignisse zu löschen sind. Dies ist darauf zurückzuführen, dass bei Beschaltung des Motors eine externe Winkeländerung nicht möglich ist.

Bei dem Einfügen von Ereignissen spielt neben der verwaltenden Liste ein weiterer Parameter eine wichtige Rolle. Dieser Parameter speichert die Endposition des Motors, die nach Abarbeitung der Liste erreicht wird. Die Aufgabe des Parameters liegt darin, während des Einfügens zu überprüfen, ob gegebenenfalls ein Endschalter erreicht wird. Ist dies der Fall müssen die Ereignisse entsprechend gekürzt werden, um ein Hinauslaufen über die Endschalter zu verhindern.

Um die Beschleunigung des Motors und die damit verbundenen Drehmomente zu beachten, ist es notwendig, die Geschwindigkeit vor dem Einfügen eines Ereignisses zu kennen. Daher muss diese berechnet werden. Sind keine weiteren Ereignisse in der Liste eingetragen, ist dies nicht nötig, da der Motor sich nicht bewegt.

Während ein Ereignis in die Liste eingefügt wird, muss der Zeitpunkt, zu dem die aus dem Ereignis resultierende Bewegung abgeschlossen ist, berechnet werden. Dieser wird benötigt, um nach Verstreichen dieser Zeit und somit zum Ende der Bewegung durch eine Nachricht einen möglichen Einrastvorgang auszulösen. Dazu wird der Motorwinkel mit der Position der Rasten auf dem Element verglichen. Befindet sich der Motor an einer der Rasten, so wird eine entsprechende externe Motorbewegung generiert und in die Liste eingefügt. Dies kann jedoch nur geschehen, wenn der Motor entsprechend stromlos geschaltet wurde.

Wird die Motorbewegung durch die Elektronik gestoppt, so muss die Liste der Ereignisse bis zu dem aktuellen Zeitpunkt abgearbeitet werden. Danach darf ihr Inhalt gelöscht werden. Außerdem müssen die auf das Ende der jeweiligen Motorbewegung wartenden Nachrichten invalidiert werden. Gegebenenfalls muss nach dem Anhalten des Motors ein Einrastvorgang ausgelöst werden.

Nachdem das Grundkonzept der Simulation beschrieben wurde, wird auf die Emulation der Firmware eingegangen.

3.2 Emulation der Elektronik

Die Emulation der Elektronik hat zwei elementare Aufgaben zu erfüllen. Zum einen muss sie wie die reelle Steuerelektronik die entsprechenden Bewegungen für die simulierten Schrittmo-

toren erzeugen, zum anderen ist sie dafür zuständig, dass die Kommunikation des realen Lucifer Instruments nachempfunden wird. Als erstes wird die Schnittstelle zwischen der nachzubildenden Elektronik und der Simulation des Instruments betrachtet.

3.2.1 Nachbildung der Steuerelektronik

Damit die Kommandos, die von der Steuerelektronik empfangen werden, richtig auf die Simulation umgesetzt werden können, ist es notwendig, dass die Steuerelektronik entsprechend nachempfunden wird. Dazu müssen sämtliche Parameter, die auch bei der reellen Steuerelektronik über Kommandos einstellbar sind, nachgebildet werden. Neben dem Speichern der Werte muss dafür gesorgt werden, dass die Parameter, die die Simulation beeinflussen, entsprechend berücksichtigt werden. Zusätzlich müssen von der nachgebildeten Steuerelektronik die entsprechenden Fahrprofile errechnet und an die Simulation weitergeleitet werden. Da der Informationsfluss nicht in Richtung der Simulation beschränkt ist, müssen außerdem von der Simulation erzeugte Nachrichten von der Steuerelektronik behandelt werden.

Als erstes werden die von der Steuerelektronik nachzubildenden Parameter näher betrachtet.

Parameter

Zum einen müssen die Parameter, die das Fahrprofil der Schrittmotoren beschreiben, nachgebildet werden. Diese wurden bereits in Kapitel 2 vorgestellt. Neben den für die Erzeugung der Fahrprofile notwendigen Parametern sind folgende für jeden einzelnen Schrittmotor geltenden Werte zusätzlich nachzubilden und in der Ansteuerung der Simulation zu beachten:

- *Übertragungsverhältnis zwischen Element und Motor*
  Dieser Parameter stellt das Übertragungsverhältnis zwischen Motor und Element dar. Dieses Verhältnis wird von der Steuerelektronik dazu benötigt, das Element um einen gegebenen Winkel zu verfahren (vergleiche das „*Go n Degrees*“ Kommando aus Anhang C).

- *Schritte pro Motorumdrehung*
  Dieser Wert wird ebenfalls benötigt, um Bewegungsangaben in Grad machen zu können. Durch ihn kann direkt auf die Schrittgröße geschlossen werden.

- *Kalibrierungsrichtung*
  Durch diesen Parameter wird die Richtung, in der die Suche nach dem Kalibrierungsschalter startet, festgelegt. Die Kalibrierungsmethode wurde bereits in Unterkapitel 2.2.2 beschrieben.

- *Kalibrierungsgeschwindigkeit*
  Mit der Kalibrierungsgeschwindigkeit wird die für die Suche nach dem Referenzschalter genutzte Geschwindigkeit vorgegeben.

- *Rückgabe des Kalibrierungswertes*
  Wird dieser Parameter für einen Motor aktiviert, wird die Simulation dazu aufgefordert, eine entsprechende Meldung zu erzeugen. Diese Meldung muss nach Beendigung der Kalibrierungsphase von der Steuerelektronik an den Rechner weitergeleitet werden. Dabei muss sie den Wert enthalten, um den die erwartete Motorposition von der Realen vor der Kalibrierung abgewichen war.

- *Position des Kalibrierungsschalters*
  Hiermit wird der Steuerelektronik mitgeteilt, ob es einen separaten Kalibrierungsschalter gibt bzw. ob ein positiver oder negativer Endschalter zur Kalibrierung zu nutzen ist.

- *Existenz von Endschaltern*
  Dieser Parameter dient dazu, die Steuerelektronik über die Existenz der Endschalter zu informieren. Werden keine Endschalter angegeben, so überwacht die Steuerelektronik auch nicht die entsprechenden Signalleitungen. Trotzdem sind die Schrittmotoren so geschaltet, dass bei Erreichen der Endschalter die Bewegung endet. Da die Steuerelektronik jedoch nicht über diese Kollision informiert wird, können auch keine entsprechenden Kollisionsnachrichten erzeugt werden.

- *Stromlosschaltung der Motoren*
  Die Stromlosschaltung der Motoren bewirkt, dass nach Beendigung einer Bewegung der Motor frei gedreht werden kann (vergleiche Kapitel 2).

Neben den schrittmotorgebundenen Parametern existieren andere, die das allgemeine Verhalten der Steuerelektronik beeinflussen. Diese werden im Folgenden aufgelistet:

- *Bewegungsbestätigungen*
  Hiermit kann beeinflusst werden, ob die Steuerelektronik nach einer Bewegung eine Bestätigung zurückliefert.

- *Mikroschrittanpassung der Bewegung*
  Dieser Parameter dient dazu, die Anzahl der in einem Fahrbefehl angegebenen Schritte entsprechend der Anzahl von Mikroschritten pro Vollschritt zu korrigieren. Somit kann bestimmt werden, ob die angegebene Schrittzahl sich auf Vollschritte oder Mikroschritte bezieht.

- *Mikroschrittanpassung der Positionsrückgaben*
  Genau wie der vorangegangene Parameter dient auch dieser der Anpassung der Mikroschritte. Statt jedoch für die Eingaben wirksam zu sein, beeinflusst er die Positionsrückgabe. Damit kann selektiert werden, ob die zurückgegebene Position in Mikroschritt oder Vollschritt gegeben ist.

- *Zeitbasis*
  Mit der Zeitbasis kann selektiert werden bezüglich welcher Maßeinheit die für die Erzeugung der Fahrprofile wichtigen Parameter angegeben werden. Zur Auswahl stehen zum einen die Sekunde, zum anderen die Zeit, die ein Zyklus der Steuerelektronik benötigt. Die Zeit eines Zyklusses beträgt ca. 0.007 Sekunden.

- *Ausgabeformatierung*
  Mit diesem Parameter kann bestimmt werden, ob Ausgaben textuell aufbereitet werden sollen oder nicht.

Nachdem alle Parameter, die für die Nachbildung der Steuerelektronik betrachtet werden, vorgestellt wurden, wird als Nächstes beschrieben, wie die emulierte Elektronik aus diesen Parametern die entsprechenden Ereignisse für das Simulationsmodell erzeugt.

Erzeugung von Bewegungen

Damit die Simulation der Schrittmotoren ordnungsgemäß durchgeführt werden kann, ist die Erzeugung der richtigen Ereignisse besonders wichtig. Dazu ist es notwendig, neben den das Profil bestimmenden Parametern die richtigen Ansteuerzeiten zu bestimmen. Da alle Fahrkommandos die zu absolvierende Strecke anstelle der Zeit vorgeben, müssen die Zeiten entsprechend berechnet werden. Diese Ansteuerzeiten werden durch die in Abb. 3.1 gezeigten Zeitintervalle

gegeben. Dazu müssen die Winkeländerungen ermittelt werden, die durch die Geschwindigkeiten aus Formel 3.4 gegeben sind. Zunächst wird die Winkeländerung berechnet, die während der Beschleunigungsphase, die zum Erreichen der gewünschten Geschwindigkeit v_{des} nötig ist, zurückgelegt wird. Dazu wird als erstes die Zeit zum Erreichen der gewünschten maximalen Beschleunigung a_{des} berechnet. Aus

$$signum(\dot{a}) = \sigma = \begin{cases} 1 & , a \leq a_{des} \\ -1 & , a > a_{des} \end{cases} \tag{3.5}$$

folgt, dass die Zeit $t_{a_{des}}$ zum Erreichen der Beschleunigung a_{des} gegeben wird durch:

$$t_{a_{des}} = \frac{a_{des} - a}{\sigma \dot{a}} \tag{3.6}$$

Die in dieser Zeit erreichte Geschwindigkeitssteigerung $v_{a_{des}}$ erhält man durch Einsetzen in Formel 3.1.

$$v_{a_{des}} = a t_{a_{des}} + \frac{\sigma \dot{a} t_{a_{des}}^2}{2} \tag{3.7}$$

Daraus lässt sich die Zeit t_{lin} die für die lineare Beschleunigung genutzt wird, mit folgender Formel berechnen:

$$t_{lin} = \frac{|v_{des} - v_0| - 2v_{a_{des}}}{a_{des}} \tag{3.8}$$

Wenn $t_{lin} < 0$ wird die gewünschte Beschleunigung a_{des} nicht erreicht, somit wird $t_{lin} = 0$ gesetzt. Damit die gewünschte Geschwindigkeit v_{des} nicht überschritten wird, muss die Zeit t_{acc} für die Beschleunigungsphase kleiner $t_{a_{des}}$ gewählt werden. Einen Wert für t_{acc} erhält man, wenn man die aus 3.7 folgende Gleichung nach t auflöst:

$$\frac{\sigma \dot{a} t^2}{2} + at = \frac{|v_{des} - v_0|}{2} \tag{3.9}$$

Wird dieser so berechnete Wert von t_{acc} in Gleichung 3.7 eingesetzt, erhält man eine entsprechende Geschwindigkeitssteigerung v_{acc} während der Beschleunigungsphase. Die erreichte Beschleunigung a_{acc} nach t_{acc} wird durch Einsetzen in Gleichung 3.3 bestimmt.

$$a_{acc} = a + \sigma \dot{a} t_{acc} \tag{3.10}$$

Unter Verwendung der Formel 3.2 lässt sich die während der Beschleunigungsphase bewirkte Winkeländerung $\omega_{t_{acc}}$ berechnen.

$$\omega_{t_{acc}} = \frac{a t_{acc}^2}{2} + \frac{a_{acc} t_{acc}^2}{2} + \frac{a_{des} t_{lin}^2}{2} + v_{acc}(t_{acc} + t_{lin}) + t_{lin} a_{des} t_{acc} + (2t_{acc} + t_{lin}) * \begin{cases} v_{des} & , v_0 > v_{des} \\ v_0 & , v_0 \leq v_{des} \end{cases} \tag{3.11}$$

Ist die durch die Beschleunigungs- und Abbremsphase bewirkte Winkeländerung $2\omega_{t_{acc}}$ geringer als die Gewünschte ω_{des}, so kann die Restzeit t_{rest}, die während der Plateauphase des Profils verbracht wird, leicht berechnet werden. Beträgt der Wert von $2\omega_{t_{acc}}$ mehr als die durchzuführende Winkeländerung ω_{des}, wird keine Plateauphase benötigt. Stattdessen wird versucht, durch Verkleinerung der linearen Beschleunigungszeit t_{lin} die durchzuführende Winkeländerung zu erreichen. Dazu muss die folgende Gleichung nach t umgestellt und gelöst werden.

$$\frac{a_{des}t^2}{2 * \begin{cases} 1 & , v_0 \leq v_{des} \\ -1 & , v_0 > v_{des} \end{cases}} + t * \begin{cases} v_0 + v_{acc} + a_{des}t_{acc} & , v_0 \leq v_{des} \\ t_{acc} * \left(a - a_{des} + \frac{\sigma \dot{a} t_{acc}}{2}\right) + v_{des} + t_{lin}a_{des} & , v_0 > v_{des} \end{cases}$$

$$= -\frac{\omega_{des}}{2} + \begin{cases} v_{acc}t_{acc} + \frac{(a+a_{acc})*t_{acc}^2}{2} + 2t_{acc} * \begin{cases} v_{des} & , v_0 > v_{des} \\ v_0 & , v_0 \leq v_{des} \end{cases} & , v_0 \leq v_{des} \\ 2t_{acc} * \left(v_{des} + at_{acc} + \frac{\sigma \dot{a} t_{acc}^2}{2} + t_{lin}a_{des}\right) & , v_0 > v_{des} \end{cases} \tag{3.12}$$

Befindet sich der so neu berechnete Wert in einem Intervall von $[0, t_{lin}]$ war die Verkleinerung von t_{lin} erfolgreich. Wenn nicht, muss $t_{lin} = 0$ gesetzt und t_{acc} angepasst werden. Dazu muss die folgende Gleichung nach t umgestellt und gelöst werden:

$$\frac{\sigma \dot{a} t^3}{2} + at^2 + (v_0 t - \frac{\omega_{des}}{4}) * \begin{cases} 1 & , v_0 \leq v_{des} \\ -1 & , v_0 > v_{des} \end{cases} = 0 \tag{3.13}$$

Mit den so berechneten Werten für t_{acc}, t_{lin} und t_{rest} ist es nun möglich, die entsprechenden Ereignisse für die Simulation zu erzeugen. Dabei haben die Intervalle $[t_0, t_1]$, $[t_2, t_3]$, $[t_4, t_5]$ und $[t_6, t_7]$ eine Länge von t_{acc}, die Intervalle $[t_1, t_2]$ und $[t_5, t_6]$ eine Länge von t_{lin} und das Plateauintervall $[t_3, t_4]$ eine Länge von t_{rest} (vergleiche Abb. 3.1). Da die Zeiten für die vorgegebenen Parameter berechnet wurden, müssen somit nur die unterschiedlichen Startgeschwindigkeiten der Profilsegmente nach Formel 3.4 berechnet werden.

Die Berechnungen der Zeiten wurden exemplarisch an dem S-Kurvenprofil vorgenommen, welche eine Verallgemeinerung der anderen beiden Profile darstellt. Durch einfaches Nullsetzen der entsprechenden Parameter erhält man die Intervallgrenzen der anderen Profile.

Neben der Erzeugung der Fahrprofile und der Verwaltung der Elektronikparameter ist die Steuerelektronik für die Generierung der Nachrichten zuständig.

Nachrichten

Da die nachgebildete Elektronik in direktem Kontakt zu der Simulation steht, ist sie für das Bearbeiten der Rückmeldungen der Simulation verantwortlich. Dazu werden simultan zur Erzeugung der Bewegungsereignisse Nachrichten generiert. Da die Simulation bei dem Einfügen von neuen Ereignissen die voraussichtliche Endposition sowie die Zeit berechnet, die der Schrittmotor benötigt, um diese Position zu erreichen, müssen die entsprechenden Nachrichten erst nach Abblauf dieser Zeit erzeugt werden. Außerdem kann die Simulation schon vorausberechnen, ob ein Endschalter erreicht wird oder nicht. Somit sind der Zeitpunkt der Ausgabe einer Nachricht sowie ihr Inhalt schon im Voraus zu bestimmen. Nachdem sie generiert worden sind, werden sie inaktiviert, um nach dem bereits bekannten Zeitintervall in Erscheinung zu treten.

Wird die Simulation während dieser Wartephase angehalten, wie z.B. durch den *„Motor Stop“* Befehl aus Anhang C, so müssen alle wartenden Nachrichten gelöscht werden, da der Zeitpunkt ihres Eintretens von der Simulation nicht erreicht wird.

Durch Umsetzung der genannten Punkte wird es möglich, das Innenleben der Elektronik zu emulieren. Es werden alle wichtigen Parameter der Elektronik nachgebildet. Außerdem werden die entsprechenden Ereignisse zur Ansteuerung der Simulation sowie die daraus resultierenden Nachrichten generiert. Als zweiter wichtiger Punkt der Emulation der Elektronik bleibt die Nachbildung der Kommunikationsschnittstelle zu betrachten.

3.2.2 Kommunikation

Damit die Kommunikation mit dem virtuellen astronomischen Instrument nicht von der des Realen zu unterscheiden ist, ist es notwendig, dass die Kommunikationsschnittstelle entsprechend nachgebildet wird. Dazu müssen alle in Anhang C genannten Steuerkommandos erkannt und entsprechend von der nachgebildeten Elektronik umgesetzt werden. Da die in Anhang C befindliche Liste der Steuerkommandos aufgrund der ständigen Weiterentwicklung der realen Elektronik noch keinen festen Standard erreicht hat, ist es wichtig, den Teil der Kommandoerkennung sehr flexibel zu gestalten. Dazu ist es erforderlich, einen in Grenzen frei konfigurierbaren Parser zu entwickeln. Bevor sich diesem Parser zugewandt wird, ist es notwendig, die Kommandostruktur zu analysieren.

Kommandostruktur

Die von der vom Heidelberger Max Planck Institut für Astronomie entworfene Steuerelektronik kommuniziert über eine eigene Kommandosprache. Dabei sind die verwendeten Kommandos kontextsensitiv. Die Bedeutung eines Zeichens erschließt sich nur unter Betrachtung der vorangestellten Zeichen. Im Gegensatz dazu könnte man bei einer kontextfreien Kommandostruktur die Bedeutung an den Zeichen selbst oder ihrer Position ausmachen[3].

Die Kommandostruktur der Steuerelektronik kann durch eine linksassoziative Grammatik beschrieben werden. Eine Grammatik wird durch eine Syntax und eine Semantik definiert. Dabei beinhaltet die Semantik die Bedeutung der Zeichen. Die Syntax sagt aus, nach welchen Regeln Zeichen kombiniert werden können. Chomsky[4] hat 1956 eine Grammatik als Viertupel definiert. Dieses setzt sich aus einem endlichen Alphabet A, einer endlichen Menge von Variablen V, einem Startsymbol S und einer endlichen Menge an Produktionen P zusammen (siehe [Weg93]). Bei einer linksassoziativen Grammatik werden nur Produktionen der Form $(v \rightarrow S, v \rightarrow va, v \rightarrow a \, mit \, v \in V, a \in A)$ erlaubt (vergleiche [Hau00]).

Nachdem die Kommandostruktur formalisiert wurde, ist es möglich, einen entsprechenden Parser zu entwickeln.

Parser

Für die Entwicklung eines Kommandoparsers ist es notwendig, die aus der allgemeinen Grammatikdefinition bekannten Mengen nachzubilden. Dabei wird das Startsymbol, die Variablen sowie die Produktionen durch die logische Struktur des Parser vorgegeben. Da sich die Syntax der Kommandos durch Weiterentwicklung der Elektronik ändert, muss es möglich sein, die Syntax, die der Parser benutzt und die durch das Startsymbol, Variablen und Produktionen definiert wird, sehr flexibel anzugeben. Da die Kommandos nach einer linksassoziativen Grammatik gebildet werden, sollte die Strukturdefinition des Parsers baumartig geschehen. Abb. 3.3 zeigt diese Struktur am Beispiel des *„Set Motor Definition“* Kommandos aus Anhang C. Dabei wurde näher auf die Auswahl *„1“*, die Wahl des Fahrprofils, sowie die Auswahl *„8“*, die Bestimmung der Mikroschrittunterteilung, eingegangen. Die genauen Bedeutungen der angezeigten Werte sind Anhang C zu entnehmen.

Für eine sehr flexible Speicherung des Alphabets sowie der dazugehörigen Bedeutungen wird eine Datenstruktur benötigt. Damit ein schneller Zugriff auf die Zeichen des Alphabets möglich ist, werden diese in einer Hashtabelle abgelegt. Eine Hastabelle stellt eine Datenstruktur dar, in der Information gespeichert werden. Dabei erfolgt der Zugriff auf die einzelnen Elemente der Hashtabelle über einen eindeutig zugeordneten Schlüsselwert. Dieser Schlüsselwert

[3] Vergleiche Chomsky Sprachhierarchie in [Weg93].

[4] Linguist, der sich mit Sprachformalismen beschäftigte.

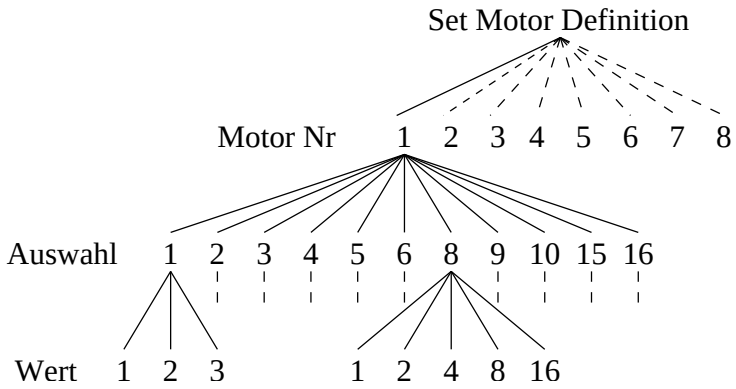


Abbildung 3.3: Beispiel Kommandostruktur

wird mittels einer Hashfunktion auf die entsprechenden Speicherbereiche abgebildet und erlaubt so einen sehr schnellen Zugriff sowie ein sehr schnelles Einfügen. Erklärungen zu Hashtabellen sowie Beispiele für Hashfunktionen finden sich in [Knu98b] und [Güt92].
Nach der Beschreibung der Emulation der Elektronik, wird auf die Visualisierung des virtuellen astronomischen Instruments eingegangen.

3.3 Visualisierung

Als letzter der drei Aspekte des virtuellen Instruments ist die Visualisierung zu betrachten. Sinn und Zweck einer guten Visualisierung besteht darin, dem späteren Benutzer auf möglichst einfache Weise eine - unter Umständen auch komplexe - Datenmenge zu präsentieren.

Bei der Visualisierung ist darauf zu achten, dass die Simulation der Mechanik und die Emulation der Firmware nicht durch die Visualisierung gestört werden. Die Visualisierung darf in keiner Weise die Werte der Simulation der Mechanik oder der Emulation der Firmware verändern. Sie darf diese nur für den Benutzer aufbereiten und präsentieren.

Laut [Str98] sind diese Datenmengen typischerweise sehr umfassend, hoch komplex und manchmal sogar widersprüchlich. Während der Bereich der Computergraphik sich jedoch hauptsächlich mit der Geometrie von Objekten beschäftigt hat, haben andere Gebiete sich hauptsächlich auf eine textuelle Ausgabe beschränkt. Erst in letzter Zeit wird versucht, diese beiden Ausgabeformen sinnvoll miteinander zu verbinden. Diese Verbindung der graphischen und textuellen Visualisierung wird auch für das virtuelle Instrument angestrebt. Eine graphische Visualisierung empfiehlt sich für die Form, Aussehen, Bewegung und Struktur des virtuellen Instruments. Die textuelle Visualisierung hingegen ist hilfreich für die Kommunikation sowie die Darstellung von elementaren Werten des Instruments.

3.3.1 Anforderungen an die Visualisierung

Für eine gute Visualisierung sind laut [Str98] folgende Punkte zu beachten:

„Selektion“ Unter Selektion versteht man die gezielte Auswahl von Daten. Diese Auswahl dient dem Fokussieren auf für den Benutzer wichtige Daten. Zu dieser Auswahl kommt außerdem das Weglassen von Daten. Gründe für dieses Nichtdarstellen von Informationen können sein:

- Informationen können nicht dargestellt werden, da sie von anderen Daten überlagert werden. Dies kann zum Beispiel durch eine geometrische Überlagerung bei einer graphischen Visualisierung der Fall sein.

- Die Informationen sind unwichtig. Meistens sind viele Daten für den Benutzer unwichtig, werden jedoch dargestellt. Durch eine gezielte Selektion hingegen kann der Benutzer selbst bestimmen, welche Daten ihn interessieren und welche nicht.

- Die Informationen können nicht bekannt sein und somit auch nicht dargestellt werden. So kann zum Beispiel eine Zufallsverteilung angegeben werden; es wird jedoch schwer möglich sein, die genaue nächste Zufallszahl zu spezifizieren, wenn diese von der Zeit abhängig ist.

- Die Informationen können nicht berechenbar sein. So lässt sich z.B. nicht aus der unscharfen Beschreibung "rot" für eine Wellenlänge zwischen 600nm und 770nm die genaue Wellenlänge berechnen.

- Aus Mangel an Ressourcen müssen Informationen weggelassen werden. So kann der Bereich, in dem die Daten dargestellt werden sollen, zu klein sein oder es ist aus zeitlichen Gründen nicht möglich, sämtliche Daten anzuzeigen.

„Strukturierung" Die Daten sollten wenn möglich eine ihnen inneliegende Struktur wiedergeben. Diese Struktur kann sich sowohl in der textuellen als auch in der graphischen Visualisierung wiederfinden. In der textuellen Visualisierung kann eine Struktur durch ein Sortierungskriterium gegeben sein. So können z.B. Ereignisse nach ihrer zeitlichen Reihenfolge sortiert werden. Zu einer guten Strukturierung textuell wiedergegebener Daten gehört außerdem das Hervorheben der Struktur. Farben, Schriftarten und unterschiedliche Größen spielen hierbei eine sehr wichtige Rolle. In der graphischen Visualisierung ist meistens eine Strukturierung bereits vorhanden, wenngleich sie nicht offensichtlich zu erkennen ist. Die graphischen Modelle sollten demnach ihre entsprechenden Strukturen unterstützen. Zum Beispiel ist in dem virtuellen Instrument eine Struktur durch die unterschiedlichen Baugruppen gegeben, aus denen eine Interaktion der einzelnen Elemente innerhalb einer Baugruppe hervorgeht.

„Abstraktion" Abstraktion ist der dritte und letzte Gesichtspunkt einer guten Visualisierung. Das Wort Abstraktion entstammt dem lateinischen Verb *„abstrahere"* was laut [MG83] soviel wie wegziehen, fortreißen, abziehen, trennen oder ausschließen bedeutet. Allen Übersetzungen gemeinsam ist das Entfernen einer Sache. Bei der Visualisierung stellt die Abstraktion das Weglassen von Unnötigem und das Fokussieren auf die wichtigen Informationen dar. Wie auch die Selektion dient die Abstraktion der Reduzierung der dargestellten Informationen. Im Gegensatz zu der Selektion wird die Abstraktion jedoch hauptsächlich durch den Computer vorgenommen. Abstraktion im Bereich der computererzeugten Visualisierung wird wie folgt definiert: *„a process by which an extract of an information space is refined so as to reflect the importance of the features of the underlying model for the dialog context and visualisation goal at hand."* [Str98] Durch eine gute Abstraktion sollte somit ein gewählter Informationsbereich so aufbereitet werden, dass die Wichtigkeit der einzelnen Merkmale des Datenmodells wiedergespiegelt wird. Dies kann durch Filterung von ungewünschten Informationen, Anpassung der Darstellung (Größe, Form, Position, Ausrichtung) oder das Hervorheben von Teilinformationen geschehen.

Neben den Gedanken, die man sich über die Anforderungen macht, die eine Visualisierung erfüllen soll, ist es ebenso wichtig, sich Gedanken über eine graphische Benutzungsoberfläche zu machen.

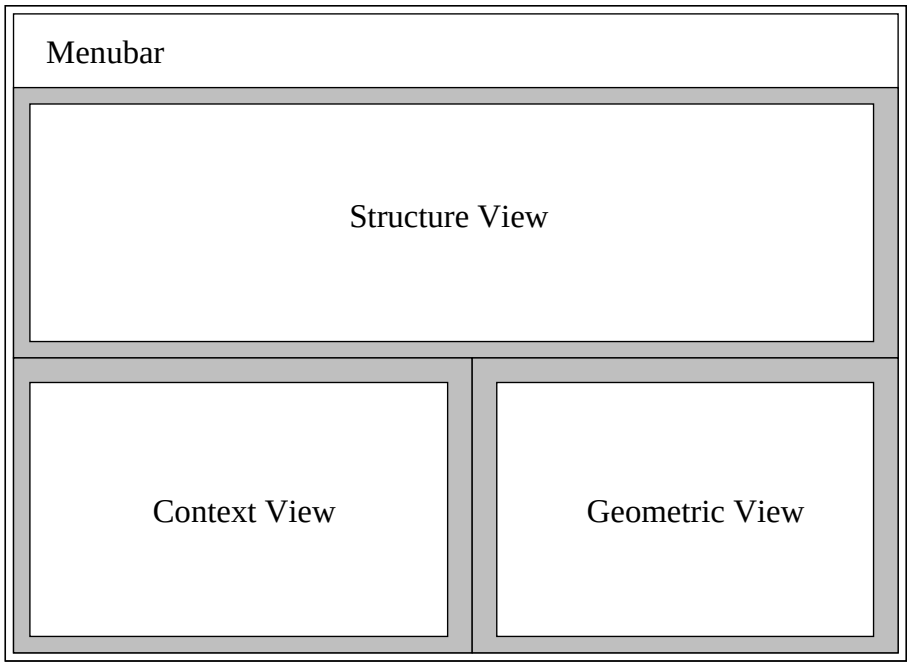


Abbildung 3.4: GUI Layout für geometrische Modelle (Quelle: [Str98])

3.3.2 Anforderungen an eine graphische Benutzungsoberfläche

Für eine graphische Benutzungsoberfläche gelten prinzipiell die gleichen Anforderungen wie für eine gute Visualisierung. Da interaktive Systeme in den letzten Jahren sehr an Komplexität zugenommen haben, ist es wichtig, Abstraktionstechniken für das Design der graphischen Benutzungsoberfläche heranzuziehen. Das Hauptaugenmerk dieser Techniken liegt bei dem Ausblenden irrelevanter Funktionen, was wiederum zu einer Vereinfachung der Bedienstruktur führen soll. Leider gibt es für die Gestaltung einer guten graphischen Benutzungsoberfläche keine allgemeinen Regeln. Von Softwarehersteller zu Softwarehersteller variieren die angewandten Designkonzepte mitunter enorm. Einzig das Konzept eines virtuellen Desktops, der Nachbildung des Arbeitsplatzes, ist fast allen Herstellern gemeinsam. Hierbei erscheinen die Benutzerdialoge als sogenannte Fenster.

Das XEROX STAR System hat auf dem Bereich der graphischen Benutzungsoberflächen Pionierarbeit geleistet. So findet man hier z.B. erstmals ein Programm, das den Benutzer auswählen lässt, in welchem Wissensstadium er sich befindet. Nur in einem sogenannten Expertenmodus stehen dem Benutzer sämtliche Funktionen zur Verfügung. Somit hat sich der Anfänger nur mit den elementaren Funktionen des Programms zu beschäftigen, bis er selbstständig entscheidet, für alle Funktionen aufnahmebereit zu sein. Mit diesem Selektionsansatz wird unnötige Komplexität, besonders für den Anfänger, vermieden. Dies alles ist genauer nachzulesen unter [JRV+89].

Ein weiterer Gesichtspunkt für eine benutzerfreundliche Schnittstelle ist ein gutes Hilfesystem. Dieses kann in Form eines gedruckten Handbuches, einer *„online Hilfe“* oder sogenannter *„tooltips“* vorhanden sein. Bei den *„tooltips“* handelt es sich um kleine Hinweistexte, die die einzelnen Funktionen eines Programms beschreiben. Sie erscheinen jeweils vor der Selektion einer Funktion. Ein gutes Hilfesystem muss sich nicht auf einen der drei genannten Punkte beschränken, sondern kann auch eine Mischung sein.

Wenn es nötig ist, Daten eines geometrischen Modells, wie es auch bei dem virtuellen Instrument der Fall ist, darzustellen, wird von [Str98] eine Dreiteilung der Benutzungsoberfläche vorgeschlagen. Siehe dazu Abb. 3.4.

- *Strukturansicht (Structure View)*
  In dieser Ansicht soll die Struktur des geometrischen Modells widergespiegelt werden. Diese Ansicht stellt einen zentralen Punkt der graphischen Benutzungsoberfläche dar, aus der Elemente für die detaillierte Kontextansicht oder geometrische Ansicht gewählt werden können.

- *Kontextansicht (Context View)*
  In der Kontextansicht erhält man detailliertere Informationen zu einem Element des geometrischen Modells. Hierbei sollte außerdem die oben genannte Möglichkeit der Datenselektion gegeben sein, um nur die von dem Benutzer gewünschten Daten zu präsentieren.

- *Geometrische Ansicht (Geometric View*)
  In dieser Ansicht soll das geometrische Modell dreidimensional angezeigt werden. Die Dreidimensionalität unterstützt hierbei unser visuelles Auffassungsvermögen, da wir Menschen auf dreidimensionales Sehen aus unserem täglichen Leben und aufgrund unserer Physis fixiert sind. Diese Ansicht sollte über die elementaren Möglichkeiten der Ansichtsveränderung verfügen. Hierzu zählen die Größenänderung, Positionsänderung und die Veränderung des Ansichtwinkels. Außerdem sollte es möglich sein, durch Selektion bestimmen zu können, welche Daten angezeigt werden.

Die gesamte graphische Benutzungsoberfläche wird hierbei in einem einzelnen Fenster dargestellt. Über eine gemeinsame Menüleiste (*Menubar*) können außerdem die jeweiligen Funktionen aufgerufen werden. Für die Umsetzung des virtuellen Instruments soll dieses Layout, entsprechend modifiziert, übernommen werden. Im nächsten Unterkapitel wird näher auf die gewählte Visualisierung und Strukturierung eingegangen.

3.3.3 Gewählte Visualisierung und Strukturierung

Für die Realisierung der Visualisierung des virtuellen Instruments, was einer der drei Kernpunkte dieser Diplomarbeit ist, wird sich von dem üblichen *„ein Programm = ein Fenster“* Design getrennt. Dabei wird jedoch nicht das Konzept der Dreiteilung in Struktur, Kontext und geometrische Ansicht verletzt. Durch die Aufsplittung der unterschiedlichen Inhalte in unterschiedliche Fenster soll dem Benutzer eine bessere Möglichkeit zur Selektion gegeben werden. Er kann somit selber wählen, welche Informationen wo angezeigt werden. Visualisiert werden müssen die Daten der folgenden zwei Gebiete:

- *Simulation der Mechanik*
  Die Simulation der Mechanik erzeugt viele, auf den ersten Blick unüberschaubar wirkende Daten. Deshalb muss die Visualisierung die Positionen und Ausrichtungen der Elemente, Aufbau des Instruments sowie verschiedenste Parameter und Simulationswerte sinnvoll und überschaubar darstellen. Durch eine gute Strukturierung der Daten und die Verwendung des oben erläuterten Konzepts ist es möglich, diese Informationen dem Benutzer zu vermitteln. Da die Mechanik des Instruments strukturiert ist, wird auch diese Struktur auf die Visualisierung übertragen. Das Instrument unterteilt sich in Baugruppen, die wiederum aus Unterbaugruppen oder einzelnen Elementen bestehen können. In der Strukturansicht des virtuellen Instruments soll dieser Aufbau vermittelt werden. Aus der Strukturansicht heraus können dann sowohl detaillierte Informationen in Form der Kontextansicht als auch eine dreidimensionale Darstellung, die geometrische Ansicht, gewählt werden. Diese Auswahl sollte für alle Elemente möglich sein. In der Kontextansicht sollen hauptsächlich textuell aufbereitete Daten dargestellt werden. Einzig und

allein die Winkelstellung, sowie die Geschwindigkeit des Motors, der das gewählte Element bewegt, müssen graphisch aufbereitet werden. Zur Aufbereitung der Winkelstellung reicht eine einfache den Winkel darstellende Graphik aus; die Geschwindigkeit hingegen sollte in einer Art Fahrtenschreiber dargestellt werden.

- *Emulation der Firmware*
  Bei der Emulation der Firmware kommt es darauf an, dass die gesendeten Daten in Abhängigkeit zu den empfangenen mit der realen Elektronik übereinstimmen. Daher ist es für die Visualisierung der Firmware wichtig, die gesendeten und empfangenen Daten darzustellen. Hierbei ist der Zeitpunkt, zu dem kommuniziert wurde, von äußerster Wichtigkeit, da die Synchronität ein wichtiges Merkmal einer guten Emulation ist. Nach den oben genannten Kriterien muss man für diesen Bereich der Visualisierung fordern: „Die Kommunikationsdaten sollen zeitlich sortiert, ihrem Inhalt entsprechend formatiert dargestellt werden und dem Benutzer sollten möglichst viele Möglichkeiten zur eigenen Selektion gegeben sein.“ Die für die Visualisierung der Simulationsdaten der Mechanik verwendete dreigeteilte Darstellung wird für die Visualisierung der Kommunikationsdaten nicht benötigt, da es sich einerseits nicht um ein geometrisches Modell handelt und andererseits die Strukturierung durch die Zeitpunkte gegeben ist.

Nachdem die theoretischen Überlegungen abgeschlossen sind, wird im nächsten Kapitel ihre Umsetzung besprochen.

Der Bauende soll nicht herumtasten und versuchen. Was stehen bleiben soll, muss recht stehen und wo nicht für die Ewigkeit doch für geraume Zeit genügen. Man mag doch immer Fehler begehen, bauen darf man keine.

Johann Wolfgang von Goethe

Kapitel 4

Umsetzung der Theorie

In diesem Kapitel wird die Umsetzung der theoretischen Vorüberlegungen beschrieben. Da die Implementierung des virtuellen Instruments in einer objektorientierten Programmiersprache erfolgen sollte, werden in diesem Kapitel UML (unified modelling language)-Diagramme zur Darstellung der Objektverknüpfungen, interner Abläufe und zeitlicher Zusammenhänge benutzt. Zu näheren Informationen über UML wird [Alh98] empfohlen.

Wie in dem Vorangegangenen findet man auch in diesem Kapitel eine Dreiteilung in Simulation, Emulation und Visualisierung. In den jeweiligen Unterkapiteln wird hauptsächlich die entsprechende Umsetzung der einzelnen Bereiche beschrieben, jedoch findet man auch die entsprechenden Hinweise auf Interaktion der einzelnen Gebiete miteinander. Die Dreiteilung spiegelt sich auch darin wider, dass drei separate Softwarepakete entwickelt wurden, die miteinander interagieren. Durch diese getrennten Softwarepakete wird es leichter möglich, Teile des virtuellen Instruments zu ersetzen oder in anderen Projekten wiederzuverwenden. Siehe dazu Abb. 4.1, in der die Paketstruktur dargestellt ist. Das Softwarepaket *„instrument“* ist für die Simulation der Mechanik, das *„communication“* Paket für die Emulation der Firmware und das *„visualisation“* Paket für die Darstellung der Daten sowie die Benutzerinteraktion gedacht.

Abbildung 4.1: Softwarepakete des virtuellen Instrumentes

Diese drei Softwarepakete werden von der zentralen „luciferVR“ Klasse aus initialisiert und benutzt. Auf den genauen Aufbau der Softwarepakete wird in den folgenden Unterkapiteln Bezug genommen. Nach einer Begründung der für die Umsetzung gewählten Programmiersprache steht die Umsetzung der Simulation der Mechanik an, die in Form des *„instrument“* Pakets geschehen ist.

4.1 Java

Für die Umsetzung der bei den theoretischen Überlegungen gewonnenen Erkenntnisse wird die objektorientierte Programmiersprache Java ausgewählt. Es gibt eine Reihe von Gründen, die für die Benutzung von Java sprechen. Laut der Firma Sun ist Java „eine einfache, objektorientier-

te, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, hochleistungsfähige, Multithread-fähige und dynamische Sprache“ (frei übersetzt in [Fla98]).

4.1.1 Eigenschaften von Java

Im folgenden wird auf einige der genannten Vorteile, wie sie auch in [Fla98] beschrieben sind, eingegangen.

- *einfach*
  Mit Java wurde versucht, eine einfache, objektorientierte Sprache zu entwickeln. Deshalb wurde die vielen Programmierern bekannte C++ Notation zum größten Teil übernommen. Um den Umfang der Sprache klein und übersichtlich zu halten, wurden einige Konstrukte aus C++ weggelassen. So findet man weder eine *„goto“* Anweisung noch die aus C bekannten *„struct“* und *„union“* Konstrukte.
  Die wohl wichtigste Vereinfachung im Vergleich zu C++ stellt jedoch das Wegfallen der Zeiger und die damit verbundene Speicherverwaltung dar. In Java werden generell alle Objekte als Referenzen übergeben. Das Löschen dieser Objekte aus dem Arbeitspeicher übernimmt die automatisierte *„garbage-collection“*, die im Hintergrund wartet, bis ein Objekt nicht mehr benötigt wird. Diese Eigenschaft von Java wissen viele Programmierer, die C++ kennen, zu schätzen, da ein aufwendiges Speichermanagement nicht nötig ist.

- *objektorientiert*
  Im Gegensatz zu dem prozeduralen Programmierparadigma verlangt das objektorientierte Programmieren eine ganz andere Entwurfsphilosophie. In einem objektorientierten System werden die einzelnen Objekte nach Vorgaben, die in Form von Klassendefinitionen vorliegen, instanziert, d.h. erzeugt, und mit gegebenen Wertigkeiten belegt. In einer Klassendefinition befinden sich die Attribute eines Objekts dieser Klasse sowie die Methoden zur Veränderung dieser Attribute. So wird auch laut [DD96] ein Objekt wie folgt definiert: „Ein Objekt ist eine Entität, die durch ihren Zustand und die Manipulationsmöglichkeiten darauf charakterisiert wird.“
  Obwohl Java von seinem Aussehen sehr C++ ähnelt, handelt es sich hierbei um eine ganz andere Sprache, bei der z.B. gleiche Sprachkonstrukte ein ganz anderes Verhalten aufweisen. Java wurde von Anfang an rein objektorientiert ausgelegt, so dass sämtliche Dinge in Java, mit Ausnahme einiger primitiver Datentypen, Objekte sind. Bei C++ hingegen wurde das rein prozedurale C nur um objektorientierte Konstrukte erweitert.
  Um die Objektorientierung einer Programmiersprache richtig auszunutzen, bedarf es sehr viel Erfahrung. Es ist nicht ausreichend, nur Klassen zu verwenden, um das Programm in mehrere Dateien aufzuspalten und weiterhin prozedural zu programmieren. Vielmehr ist es von elementarer Bedeutung, das objektorientierte Programmierkonzept über Jahre hinweg durch ständiges Anwenden zu erlernen.

- *interpretiert*
  Bei Java handelt es sich um eine interpretierte Sprache. Dabei wird der Java-Quellcode in den sogenannten Byte-Code übersetzt. Dieser Byte-Code ähnelt dem architekturbezogenen Maschinen-Code mit der Ausnahme, dass er nicht eine spezielle Rechnerplattform benötigt um ausgeführt zu werden, sondern innerhalb einer virtuellen Maschine läuft. Durch Verwendung dieser virtuellen Maschine fällt das aus anderen Programmiersprachen bekannte *„linken“* , das Einbinden externer Softwarepakete, weg. Benötigt man externe Softwarepakete, so werden diese zur Laufzeit in die virtuelle Maschine geladen.

- *architekturneutral und portabel*
  Durch den bereits beschriebenen Prozess der Byte-Code-Erstellung erhält man außerdem eine auf vielen Systemen lauffähige Software. Die Software muss nicht mehr plattformabhängig erstellt werden, wobei spezielle Dinge wie Zahlenformate oder Speicherzugriffsarten zu beachten sind. Vielmehr ist es nur notwendig, eine virtuelle Maschine zu installieren. Mittlerweile findet man diese virtuellen Maschinen in Handys, Waschmaschinen oder Kaffeeautomaten, um nur einige „exotische“ Beispiele zu nennen. Diese Umsetzung des von Sun propagierten Mottos *„write once, run anywhere“* (einmal schreiben, überall ausführen) erleichtert das Erstellen von architekturneutraler Software.

- *dynamisch und verteilt*
  Das Dynamische an Java ist seine Möglichkeit, zur Laufzeit beliebige Klassen nachzuladen und in ein laufendes System einzubinden. Maschinenspezifische Codebibliotheken können ebenfalls dynamisch geladen werden. Da bei Java von Anfang an der Einsatz in Rechnernetzen mit vorgesehen wurde, bringt Java von Hause aus schon eine Vielzahl von Möglichkeiten zur Kommunikation über Netzwerke mit. Dazu gehört zum einen das RMI API (remote method invocation application program interface), zum anderen die Methode der „stream“ bzw. „socket“ orientierten Kommunikation. Nähere Informationen zu diesen Kommunikationsarten findet man in [Com01]. Ein interessantes Beispiel, das die Komplexität des Erstellens einer Netzwerkverbindung in C++ im Vergleich zum Erstellen einer Netzwerkverbindung unter Java zeigt, findet man in [Dar01].

- *robust*
  Die Java Entwickler haben viele Dinge unternommen, damit man mit Java sehr zuverlässige und robuste Software entwickeln kann. Dies erspart natürlich nicht den üblichen Prozess der Qualitätssicherung, eliminiert jedoch eine Menge von potentiellen Gefahren. Als erstes wäre hierbei das Fehlen von Zeigern zu nennen. Die einzelnen Objekte werden generell als Referenzen übergeben. Somit spart man sich den Bereich der Zeigerarithmetik, der in C++ sehr fehleranfällig ist. Bei Zugriffen auf Felder von Objekt- oder Zeichenketten werden zur Laufzeit die Bereichsgrenzen überprüft. Viele der Angriffe auf Rechner über das Internet machen sich dieses Fehlen bei C++ Programmen zu Nutze. Genauso können Stapelüberläufe unter Java nicht als Sicherheitslücke ausgenutzt werden. Das Umtypisieren von Objekten eines Typs in Objekte eines anderen wird ebenfalls während der Laufzeit durchgeführt.
  Als letzter Punkt, der eine extreme Robustheit von Java Programmen garantiert, ist noch das *„exception handling“* zu nennen. Hierbei kann der Programmierer selbst Ausnahmebedingungen definieren und festlegen, was passieren soll, wenn diese eintreten.

- *sicher*
  Da Java von Anfang an für den Einsatz in Netzen gedacht war, ist die Sicherheit der Programmiersprache ein weiterer wichtiger Aspekt. Die Sicherheit von Java beruht auf vier Eigenschaften. Als erstes ist hierbei die eben beschriebene Robustheit zu nennen, die dafür sorgt, dass kein Programm Zugriff auf nicht dafür vorgesehene Speicherbereiche bekommt. Die zweite Schutzmaßnahme von Java liegt darin, dass der Byte-Code vor seiner Ausführung überprüft wird und somit sichergestellt ist, dass er keinen illegalen Byte-Code enthält oder zu Überläufen eines Prozessstapels führt. Die dritte Eigenschaft, die zur Sicherheit von Java dient, ist die Möglichkeit, den Byte-Code mit digitalen Signaturen zu versehen, mit denen die Herkunft und die Unversehrtheit des Byte-Codes nachgewiesen werden kann. Java ermöglicht es, Programme in einem sogenannten *„Sandkasten“* auszuführen. Hiermit ist eine Umgebung gemeint, für die sämtliche Programmrechte, wie z.B. Festplattenzugriffe, vorgegeben werden können.

- *hochleistungsfähig*
  Obwohl es sich bei Java um eine interpretierte Sprache handelt, ist Java sehr leistungsfähig. Der Geschwindigkeitsunterschied zwischen Java und in plattformabhängigen Maschinencode übersetzten C++ Programmen liegt zwischen 5-facher und 10-facher Geschwindigkeit. Wird jedoch der Byte-Code durch den JIT-Compiler (just in time) während der Laufzeit in Maschinencode übersetzt, liegen, laut einer Aussage von Sun, sowohl C++ als auch Java fast gleich auf. Hinzu kommt, dass ein Großteil von Operationen wie z.B. Zeichenkettenvergleiche direkt in der virtuellen Maschine enthalten und somit in höchst effektivem Maschinencode implementiert sind. Bei den meisten GUI Programmen oder Programmen, die auf Daten aus dem Netzwerk warten, ist der übrig bleibende Geschwindigkeitsunterschied annähernd Null und somit nicht entscheidend.

- *Multithread-fähig*
  Das letzte zu behandelnde Schlagwort ist die Multithread-Fähigkeit. Java erlaubt es relativ einfach, mehrere eigenständige Prozesse gleichzeitig ablaufen zu lassen. An dieser Stelle sei wiederum auf den Vergleich zu C++ verwiesen, um die Eleganz der Threadprogrammierung unter Java zu verdeutlichen. Java bringt neben einer einfachen Prozesserzeugung auch Konstrukte mit, die es vereinfachen, Prozesse miteinander interagieren zu lassen und gegebenenfalls zu synchronisieren. Hinzu kommt, dass sich der Programmierer nicht mit der plattformabhängigen Prozessverwaltung zu beschäftigen hat, da dies alles von der virtuellen Maschine zur Verfügung gestellt wird.

Nachdem eine Menge Eigenschaften von Java beschrieben worden sind, werden im nächsten Unterkapitel die Gründe aufgelistet, die zur Entscheidung für den Einsatz dieser Programmiersprache führten.

4.1.2 Gründe für die Wahl von Java

Neben den vorgestellten Schlagworten sind vor allem folgende Punkte entscheidend, Java als Programmiersprache für das virtuelle astronomische Instrument zu verwenden.

- *umfangreich*
  Java ist von Hause aus mit einer riesigen Menge an Softwarepaketen ausgestattet. Die große Anzahl an frei nutzbaren und effektiv implementierten Paketen erspart es, während der Programmentwicklung das *„Rad erneut zu erfinden“*. So findet man nicht nur Standard Datenstrukturen vor, sondern auch Pakete zur Netzwerkkommunikation oder zur Erzeugung von graphischen Ausgaben. Ein großer Vorteil dieser enormen Menge an Bibliotheken ist ihre meist sehr gute Dokumentation. Sun hat extra dafür das sogenannte *„javadoc“* System entworfen, welches es ermöglicht, im Quelltext enthaltene Dokumentationen in ein einheitlich formatiertes Dokument zu extrahieren. Dabei werden die Daten so aufbereitet, dass Verknüpfungen zwischen unterschiedlichen Klassen zur Verfügung gestellt werden und so ein schnelles Durchmustern der Dokumentation ermöglicht wird. Neben diesen Informationen, die unter `java.sun.com` erhältlich sind, wird [Dar01, Fla98, Fla99, LW98, Knu99] als Informationsquelle benutzt.
  Nur durch den effektiven Einsatz der durch Java mitgelieferten Bibliotheken konnte das virtuelle Instrument in etwas mehr als zehntausend Programmzeilen realisiert werden. Ohne diese Bibliotheken wäre alleine die dreidimensionale Darstellung verantwortlich für 10000 Zeilen.

- *Java3d*
  Sun bietet mit der Java3d-API ein Softwarepaket an, mit dem man sehr leistungsfähige 3d-Graphiken erzeugen kann. Da bereits aus den theoretischen Vorüberlegungen hervorgeht, dass eine dreidimensionale Darstellung des virtuellen Instruments benötigt wird, stellt diese sehr einfach zu nutzende Schnittstelle die ideale Lösung dar. Java3d basiert von seinen Konzepten her auf den durch OpenGL gesetzten Standard für dreidimensionale Programmierung. Nähere Informationen zu Java3d sind [SRD00] zu entnehmen.

- *Geschwindigkeit*
  Der genannte Geschwindigkeitsnachteil von Java gegenüber C++ ist zu vernachlässigen, da der Fortschritt die Leistungsfähigkeit von Rechnern innerhalb eines Jahres um ca. 50% steigert[1]. Die hiermit einhergehende immer größere Integrationsdichte von auf Siliziumbasis erstellten Halbleiterelementen wird zwar laut [Wie92] nicht ewig anhalten, aber die Rechenleistungen in zwei Jahren wird, verglichen mit der heutigen Rechenleistung, sämtliche Geschwindigkeitsnachteile des virtuellen Instruments ausgleichen.

- *Architekturvielfalt*
  Ein weiterer Vorteil von Java liegt darin, dass das virtuelle Instrument auf sehr vielen Rechnerplattformen lauffähig sein sollte. So ist z.B der im Astronomischen Institut der Uni-Bochum zur Verfügung gestellte Arbeitsplatzrechner eine Sun Workstation mit dem Solaris Betriebssystem. Die spätere Rechnerplattform des virtuellen Instruments steht jedoch noch nicht fest. Außerdem werden von den anderen am Lucifer Projekt beteiligten Instituten verschiedenste Rechner- und Betriebssysteme verwendet. Wäre versucht worden, simultan für die unterschiedlichen Hard- und Softwareumgebungen das virtuelle astronomische Instrument zu entwickeln, hätte wahrscheinlich diese Diplomarbeit nicht fertiggestellt werden können.

- *kostenlos erhältlich*
  Als letzter und aufgrund der finanziellen Lage der Universitäten wohl auch sehr wichtiger Grund bleibt die kostenlose Verfügbarkeit von Java und eine unter Java funktionierende Entwicklungsumgebung zu nennen. Sowohl für Windows als auch für Solaris Rechner sind relativ hohe Beträge für auf diese Betriebsysteme zugeschnittene C++ Compiler zu zahlen.

Nachdem eine Reihe von Gründen für die Verwendung von Java als Programmiersprache genannt worden sind, wird im nächsten Unterkapitel die verwendete Entwicklungsumgebung vorgestellt.

4.2 Entwicklungsumgebung

Neben der Wahl der richtigen Programmiersprache ist eine sehr gute Entwicklungsumgebung für die erfolgreiche Durchführung eines Softwareprojektes notwendig. Als Entwicklungsumgebung wird Together von der Firma Togethersoft gewählt (siehe `www.togethersoft.com`). Nach dem Programmstart der Together Entwicklungsumgebung erscheint das in Abb. 4.2 gezeigte Fenster, in dem man den Initialisierungsprozess der jeweiligen Werkzeuge verfolgen kann.

[1] Diese Gesetzmäßigkeit wurde erstmals von dem Intel Mitbegründer Gordon E. Moore erkannt (siehe `www.intel.com/pressroom/kits/bios/moore.htm`). Mittlerweile wird jedoch das Wachstumspotential selbst von ihm als kritisch betrachtet. (siehe [Bon03]).

Abbildung 4.2: Startbildschirm von Together

4.2.1 Gründe für die Wahl von Together

Kriterien für die Wahl von Together als Entwicklungsumgebung sind:

- *plattformunabhängig*
  Die Together Entwicklungsumgebung ist mit der Programmiersprache Java umgesetzt worden. Somit sind auch alle Vorteile von Java in ihr enthalten. Der ausschlaggebende Vorteil dabei ist die Plattformunabhängigkeit, da das virtuelle Instrument auf verschiedenen Hardwarearchitekturen lauffähig sein soll. Damit ist automatisch gefordert, dass für Wartungsarbeiten an der Software die Entwicklungsumgebung auch für die jeweilige Plattform verfügbar ist.

- *kostenlos erhältlich*
  Obwohl die Together Entwicklungsumgebung im Handel sehr teuer zu beziehen ist, besteht für Bildungseinrichtungen - durch ein spezielles Förderungskonzept - die Möglichkeit Softwarelizenzen für Together kostenlos zu erwerben.

- *UML fähig*
  Together bietet sehr gute und umfangreiche Möglichkeiten zur Planung von Softwareprojekten mittels UML. Eine Vielzahl der durch UML spezifizierten Diagrammtypen werden durch Together unterstützt. Außerdem bietet Together die Möglichkeiten, sowohl aus den UML-Diagrammen ein Programmskelett anfertigen zu lassen als auch Änderungen am Quelltext in die Diagramme zu übernehmen.

- *sehr umfangreich*
  Der Funktionsumfang von Together ist enorm. So wird neben den bereits genannten UML-Werkzeugen ein Versionsmanagement, eine Testumgebung, ein integrierter Compiler, ein Debugger, Werkzeuge zur Dokumentation, Werkzeuge zur Fehlersuche sowie ein guter Texteditor mitgeliefert. Außerdem bietet Together die Möglichkeit, verschiedenste externe Werkzeuge zu integrieren.

- *guter Editor*
  Der Editor, der das Kernstück der Programmierung darstellt, bietet alle gängigen Unterstützungsmöglichkeiten. Neben den gängigen wie Syntaxhervorhebung und automatischer Formatierung bietet er z.B. Funktionen, wie das Hervorheben von Fehlern während der Eingabe sowie die automatische Vervollständigung von Methoden- oder Attributsnamen.

Nachdem die Gründe für die Wahl der Entwicklungsumgebung erläutert worden sind, wird diese näher beschrieben.

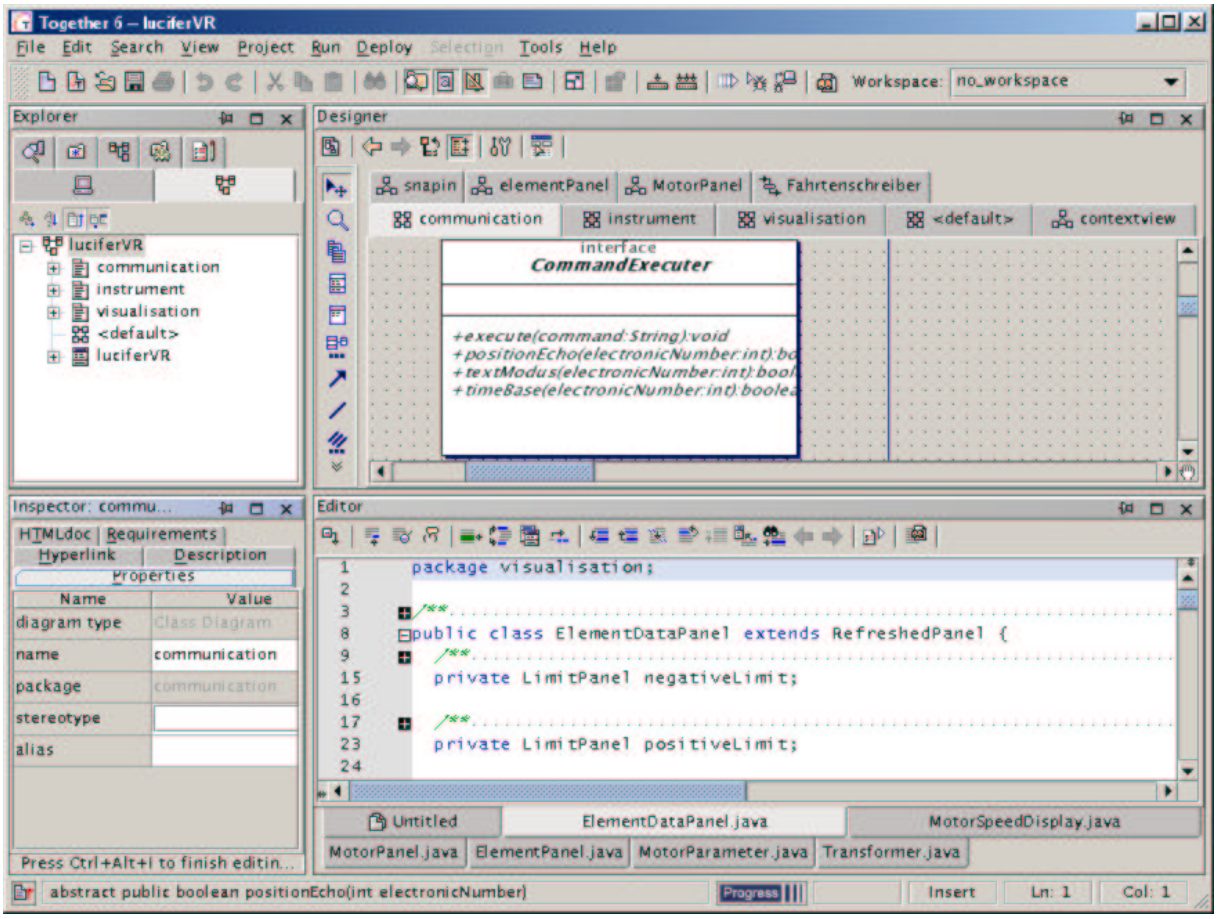


Abbildung 4.3: Benutzungsoberfläche von Together

4.2.2 Benutzungsoberfläche

Nach Initialisierung der einzelnen Werkzeuge erscheint, je nach Konfiguration, ein Abb. 4.3 ähnelndes Fenster. Aus Abb. 4.3 ist der Funktionsumfang der Entwicklungsumgebung zu erahnen, deswegen erfolgt eine Beschränkung auf die wichtigsten Elemente. Die gezeigte Benutzungsoberfläche unterteilt sich grob in fünf Bereiche.

Der erste Bereich enthält die oben gelegene Menüleiste, über die die Entwicklungsumgebung konfiguriert werden kann und die meisten Programmfunktionen erreichbar sind. Darunter befindet sich der eigentliche Arbeitsbereich von Together. Er setzt sich aus einer Strukturansicht, einer Kontextansicht, dem UML-Editor und dem Texteditor zusammen. Die Strukturansicht dient dem schnellen Zugriff auf Softwarepakete, Klassen, Methoden, Attribute und Diagramme. Die dort selektierten Elemente werden in dem entsprechenden UML-Diagramm, der Kontextansicht sowie dem Texteditor geöffnet, um dort bearbeitet zu werden. Zu näheren Informationen über Together ist `www.togethersoft.com` zu empfehlen.

Ein wichtiger Gesichtspunkt einer guten Softwareentwicklung ist, dass eine gute Programmdokumentation ermöglicht wird.

4.2.3 Durch Together erstellte Dokumentation

Together ermöglicht eine gute Dokumentation der mit dieser Entwicklungsumgebung entworfenen Programme. Zum einen stehen die aus den Vorüberlegungen hervorgegangenen UML-Diagramme als Dokumentation zur Verfügung, zum anderen erstellt Together eine sehr umfangreiche Klassendokumentation.

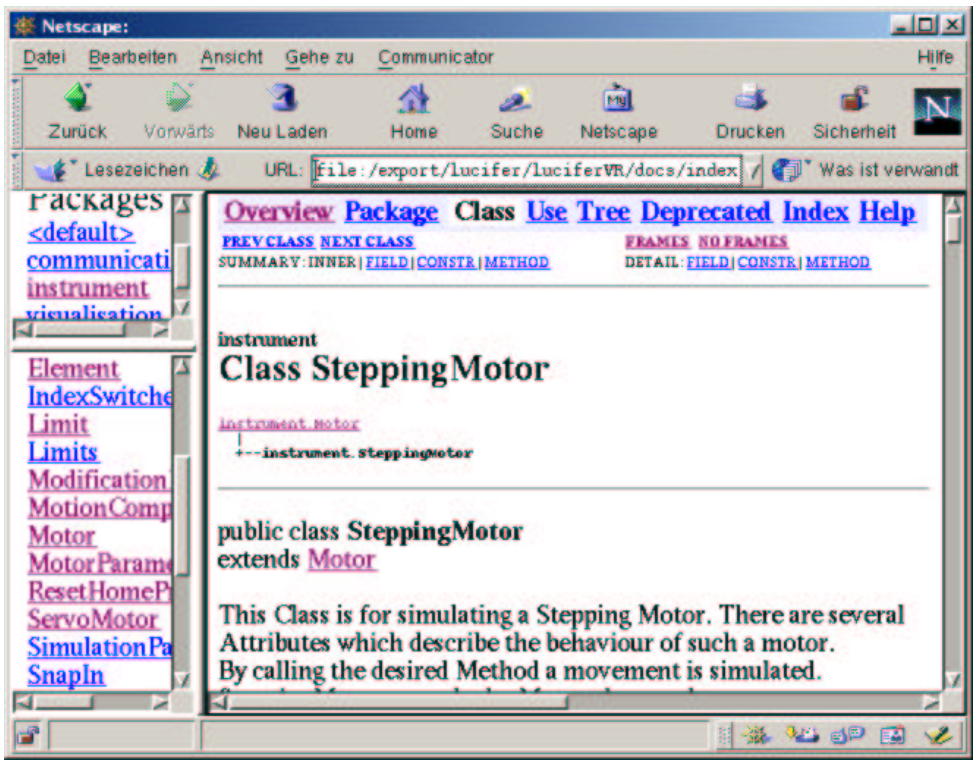


Abbildung 4.4: Dokumentation im Webbrowser

Dies geschieht dadurch, dass Together das von der Firma Sun entwickelte „*javadoc*“ System benutzt. Hierbei werden im Quelltext enthaltene Dokumentationen in verknüpfte HTML-Dateien exportiert. Als Beispiel zeigt Abb. 4.4 eine so erstellte HTML-Dokumentation an. Damit das „*javadoc*“ System eine solche Dokumentation erstellen kann, ist es nötig, im Quelltext selber speziell formatierte Informationen anzugeben. Durch Voranstellen vordefinierter Bezeichner wird dem „*javadoc*“ System mitgeteilt, welche Informationen bestimmte Sachverhalte dokumentieren. So zeigt z.B. „*@author max mustermann*“ an, dass ein spezieller Programmteil von Max Musterman geschrieben wurde oder „*@param anzahl = anzahl an übergebenen elementen*“ bedeutet, dass der Parameter „*Anzahl*“ die Anzahl der übergebenen Elemente spezifiziert.

In dem in Abb. 4.4 gezeigten Beispiel ist die Klasse „*SteppingMotor*“ dargestellt. Man kann erkennen, dass diese Klasse von der „*Motor*“ Klasse abgeleitet ist. Außerdem sieht man, dass die Klasse Bestandteil des „*instrument*“ Pakets ist. Durch Anklicken der jeweiligen Verweise bekommt man detaillierte Informationen zu den selektierten Paketen, Klassen, Methoden oder Attributen. Nähere Informationen zu dem „*javadoc*“ System sind unter `java.sun.com` zu erhalten.

Nachdem die verwendete Entwicklungsumgebung vorgestellt wurde, wird nun die Umsetzung der theoretischen Vorüberlegungen behandelt. Als erstes ist dabei die Simulation der Mechanik in Form des „*instrument*“ Pakets zu betrachten.

4.3 Simulation der Mechanik

Aufgabe der Simulation der Mechanik ist es, das zeitliche Verhalten der Komponenten im Instrumenteninneren nachzubilden. Dazu wurden in Kapitel 3 die zu simulierenden Werte festgelegt. Unter Verwendung einiger vereinfachender Annahmen wurde ein entsprechendes Simulationsmodell entwickelt. Die Umsetzung dieses Simulationsmodells mittels der Programmiersprache Java wird in diesem Unterkapitel beschrieben. Dazu wurde ein eigenes Softwarepaket

mit dem Namen „*instrument*“ angelegt. In der Simulation wird die Winkeländerung der Schrittmotoren berechnet. Diese wirkt sich auf die Positionen der einzelnen mechanischen Elemente aus. Die auf den Elementen montierten Rasten und Referenzschalter beeinflussen wiederum die Simulation der Schrittmotoren. Daher ist es wichtig die Datenstruktur, die den Aufbau der Mechanik beschreibt, näher zu betrachten.

4.3.1 Datenstruktur der Mechanik

Die Datenstruktur der Mechanik definiert neben den Motoren selbst einige für die Simulation wichtige Parameter. Wie in Abb. 4.5 dargestellt, ist die „*Motor*“ Klasse ein zentrales Element der für die Beschreibung der Mechanik genutzten Datenstruktur.

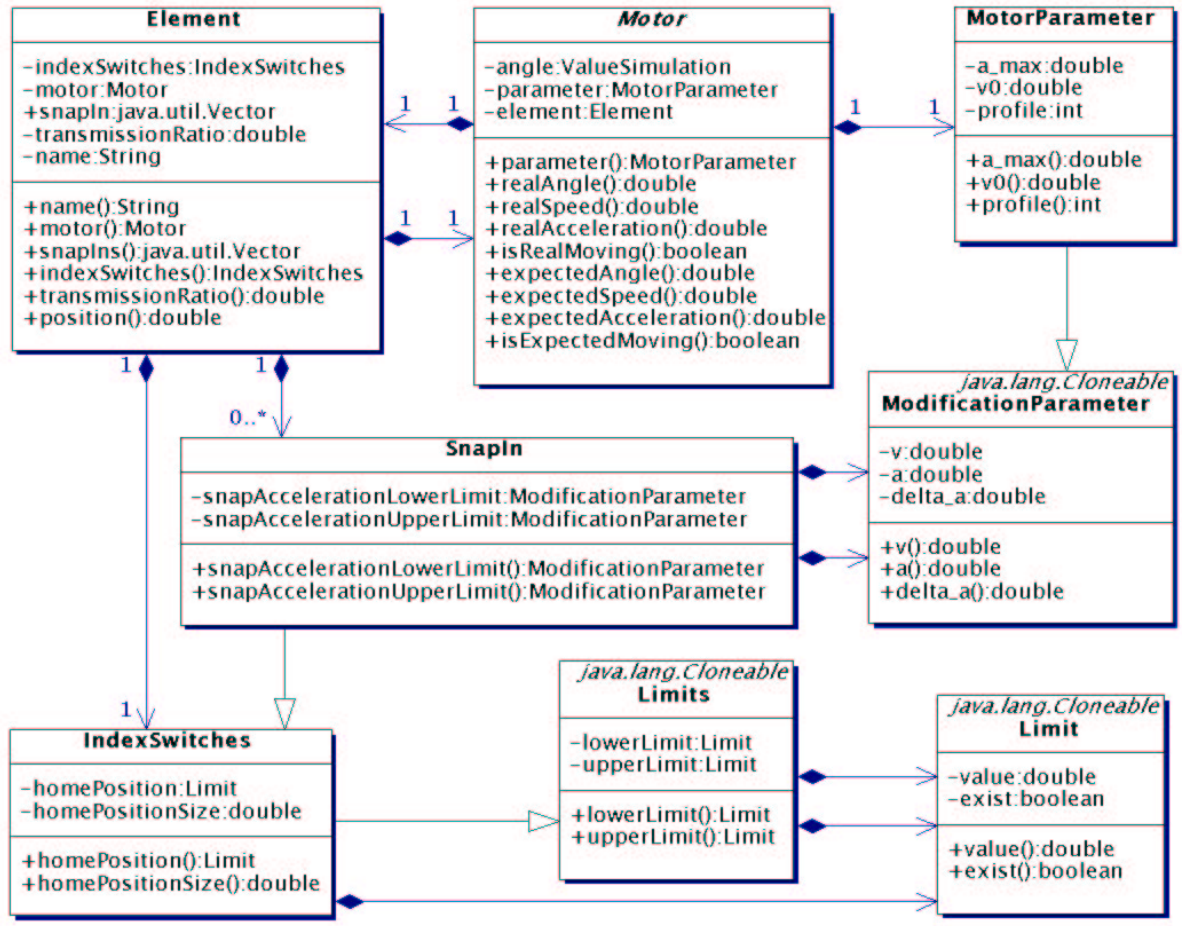


Abbildung 4.5: Datenstruktur der Simulation

Die „*Motor*“ Klasse bezieht ihre Informationen zu Winkel, Geschwindigkeit und Beschleunigung direkt aus der angeschlossenen Simulation, die in Unterkapitel 4.3.3 beschrieben wird. Um diese Werte an andere Objekte weitergeben zu können, besitzt die „*Motor*“ Klasse entsprechende Methoden sowohl für die reellen Werte als auch die erwarteten (vergleiche Kapitel 3).

Neben einer Schnittstelle zur Simulation besitzt jedes „*Motor*“ Objekt die für einen Fahrbefehl notwendigen Parameter. Dazu erbt die „*MotorParameter*“ Klasse, die selbst die Startgeschwindigkeit v_0, die gewünschte Beschleunigung a_{des} sowie das Fahrprofil definiert, von der „*ModificationParameter*“ Klasse. Diese Klasse beinhaltet einen Satz von Parametern, der für ein Ereignis der Simulation benötigt wird (siehe Formel 3.1). In Verbindung mit der „*MotorParameter*“ Klasse werden diese Parameter als gewünschte Geschwindigkeit v_{des}, Startbeschleunigung a und Beschleunigungsänderung $\dot{a}$ interpretiert.

Zwischen der *„Motor“* Klasse und dem dadurch angetriebenen Objekt der *„Element“* Klasse besteht eine, in beide Richtungen existierende, Verbindung. Dies kommt daher, dass ein Schrittmotor ein Element antreibt, genauso wie ein Element durch einen Rastvorgang die Motorausrichtung verändern kann. Neben dem Übertragungsverhältnis zwischen Schrittmotor und Element beinhaltet die *„Element“* Klasse einen Vektor von Rasten. Diese Rasten, die in Form der *„SnapIn“* Klasse implementiert sind, erben ihre Positionsinformationen von den Referenzschaltern. Dabei werden der positive und negative Endschalter als Anfang und Ende der Raste gewertet. Der Kalibrierungschalter gibt die Position an, in welche eingerastet werden soll. Um die Einrastbewegung durchführen zu können, besitzt jedes *„SnapIn“* Objekt die entsprechenden Bewegungsparameter der *„ModificationParameter“* Klasse. Jeweils einen für das Einrasten aus negativer bzw. positiver Richtung bezüglich der Einrastposition.

Die Referenzschalter, die durch die *„IndexSwitches“* Klasse implementiert werden, bringen, neben der von der *„Limits“* Klasse geerbten möglichen oberen und unteren Schranke, zusätzlich die Position des Kalibrierungsschalters sowie seine Breite mit. Dabei wird bei der Erzeugung eines *„IndexSwitches“* Objekts dafür gesorgt, dass der Kalibrierungsschalter nur innerhalb der eventuell vorgegebenen Grenzen existiert.

Bevor die Umsetzung der Winkelsimulation erläutert wird, muss die Erzeugung der Zeit sowie das Generieren von Zufallszahlen näher betrachtet werden, da beide von der Simulation benötigt werden.

4.3.2 Zeit- und Zufallserzeugung

Um die Simulation der Mechanik entsprechend durchführen zu können, ist es notwendig, die aktuelle Zeit zurückgeliefert zu bekommen. Java bietet Methoden an, mit denen die Systemzeit eines Rechners bestimmt werden kann. Leider ist die Genauigkeit der Systemzeit dabei nur auf Millisekunden beschränkt. Das kommt daher, dass manche Rechnersysteme, die von dem architekturneutralen Java unterstützt werden, nicht die Möglichkeit zu einer genaueren Zeitbestimmung besitzen. Wenn ein Schrittmotor mit 1000 Schritten pro Sekunde fährt und die Schritte dabei noch in Mikroschritte unterteilt werden, wird deutlich, dass eine Genauigkeit im Millisekundenbereich nicht ausreichend ist. Um zu verhindern, dass aufgrund der mangelnden Zeitgenauigkeit numerische Nebeneffekte auftreten, wird die Genauigkeit der Systemzeit künstlich gesteigert. Die in Abb. 4.6 gezeigte *„TimeGenerator“* Klasse ist zuständig für die Zeiterzeugung. Um die Genauigkeit der Zeit zu steigern, wird mittels einer Zufallsfunktion der entsprechende Nanosekundenanteil zufällig generiert. Dies ist deswegen erlaubt, da die Kommunikation mit der Steuerelektronik nicht zeitlich exakt bestimmt werden kann (vergleiche Kapitel 3). Ein weiterer Grund der dieses Vorgehen erlaubt ist, dass zwischen Erzeugung der Systemzeit und Beendigung der Berechnungen eine kurze und von der Rechenleistung der verwendeten Hardware abhängige Zeitspanne liegt.

Ein *„TimeGenerator“* Objekt muss die Zeiten fortlaufend erzeugen, um zu verhindern, dass Zeitpunkte aus der Vergangenheit zustande kommen. Um diese Linearität der Zeit zu gewährleisten, merkt sich ein Objekt der *„TimeGenerator“* Klasse die zuletzt erzeugte Zeit. Da Java die Systemzeit in Millisekunden seit dem 01.01.1970 00.00 Uhr GMT zurückgibt, wird zur Steigerung der Rechengenauigkeit diese Zahl, bei Erzeugung eines *„TimeGenerator“* Objekts, auf Null gesetzt. Dies geschieht dadurch, dass die berechnete Korrekturzeit jedesmal von der Systemzeit abgezogen wird, bevor der Nanosekundenanteil addiert wird. Durch das Hinzufügen eines Skalierungsfaktors kann ein Objekt der *„TimeGenerator“* Klasse eine, relativ zu der Realzeit, verlangsamte bzw. beschleunigte Zeit erzeugen. Damit wird es möglich, das virtuelle astronomische Instrument z.B. in Zeitraffer oder Zeitlupe ablaufen zu lassen.

Zur Erzeugung des Nanosekundenanteils benutzt jedes Objekt der *„TimeGenerator“* Klasse einen eigenen Zufallszahlengenerator. Dieser Zufallszahlengenerator, der durch die *„Dis-*

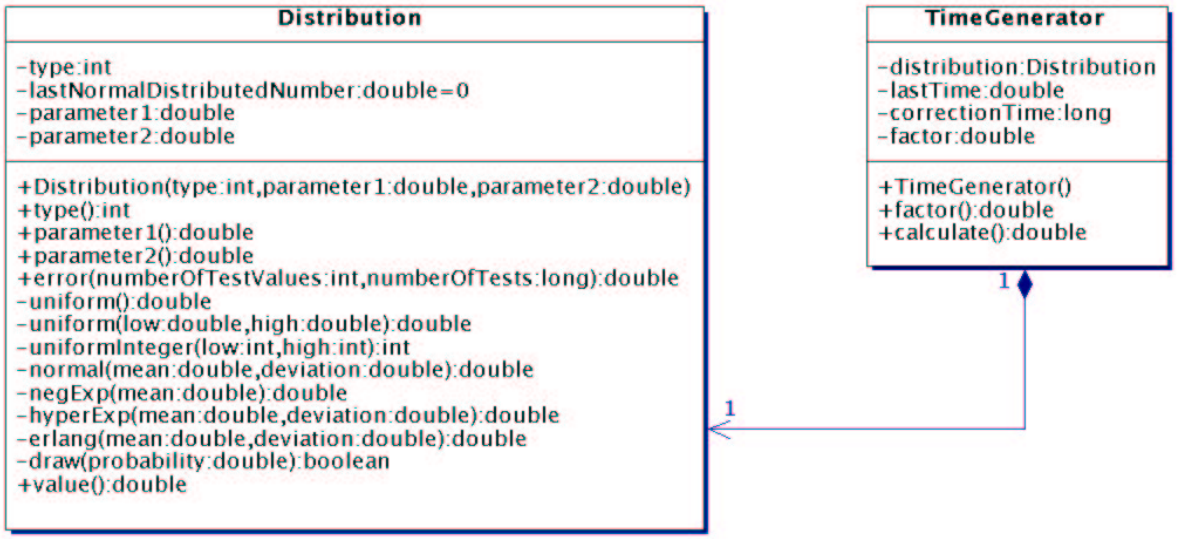


Abbildung 4.6: Klassen der Zeit- und Zufallserzeugung

tribution“ Klasse implementiert wird, benutzt die von Java gegebene Methode der uniformen Zufallszahlenerzeugung in dem Intervall [0,1[. Diese Erzeugung genügt den in [Knu98a] aufgestellten Kriterien. Die *„Distribution“* Klasse sorgt dafür, dass diese so erzeugten Zufallszahlen der angegebenen Verteilung mit den dazugehörigen Parametern folgen. Zu Auswahl stehen die uniforme Verteilung in einem vorgegebenen Intervall, für sowohl reelle als auch ganze Zahlen. Außerdem steht die Normalverteilung mit gegebenem Mittelwert und Varianz zur Verfügung. Eine negativexponentielle, eine hyperexponentielle sowie die Erlangen Verteilung werden zusätzlich implementiert. Außerdem besteht die Möglichkeit, einfache binäre Entscheidungen zu treffen. Nähere Informationen zu den Zufallsfunktionen finden sich in [BSMM99].

Als nächstes wird die Implementierung der Winkelsimulation betrachtet.

4.3.3 Winkelsimulation

Um die Umsetzung der theoretischen Vorüberlegungen zu dem Simulationsmodell aus Kapitel 3 beschreiben zu können, muss zunächst der ereignisorientierte Teil betrachtet werden. Die Bewegungsereignisse der Simulation werden durch die in Abb. 4.7 gezeigte *„ValueModifier“* Klasse nachgebildet. Die für die Berechnung eines Bewegungsereignisses notwendigen Parameter erbt die *„ValueModifier“* Klasse von der in Abb. 4.5 gezeigten *„ModificationParameter“* Klasse. Diese werden für die in Kapitel 3 vorgestellten Formeln benötigt. Implementiert sind die Formeln direkt in den entsprechenden Methoden eines Ereignisses. Diese sind statisch ausgeführt und werden somit nicht für jedes Objekt einzeln erzeugt, sondern stehen allen gemeinsam zur Verfügung. Sie berechnen jeweils für ein übergebenes *„ValueModifier“* Objekt und einen gegebenen Zeitpunkt bzw. Dauer den entsprechenden Wert. Dies kann die erreichte Beschleunigung, erreichte Geschwindigkeit oder zurückgelegte Winkeländerung sein. Die dazu verwendeten Formeln sind 3.1, 3.2 und 3.3. Bei den Berechnungen wird jeweils zwischen den reellen Motorwerten und denen die von der Steuerelektronik erwartet werden, unterschieden. Die für die Berechnung benötigten Simulationsparameter werden in Abb. 4.8 näher beschrieben.

Wird ein neues Bewegungsereignis in Form eines *„ValueModifier“* Objektes erzeugt, dann werden die entsprechenden aus dem Ereignis resultierenden Werte berechnet und in speziell dafür vorgesehenen Attributen gespeichert. Wird ein Ereignis zerteilt, da eine Berechnung bis zu einem Zeitpunkt innerhalb des Ereignisses stattfindet oder da aufgrund eines Endschalters das Ereignis nicht ganz ausgeführt werden kann, so erlaubt die *„decreaseDurationTo“* Methode

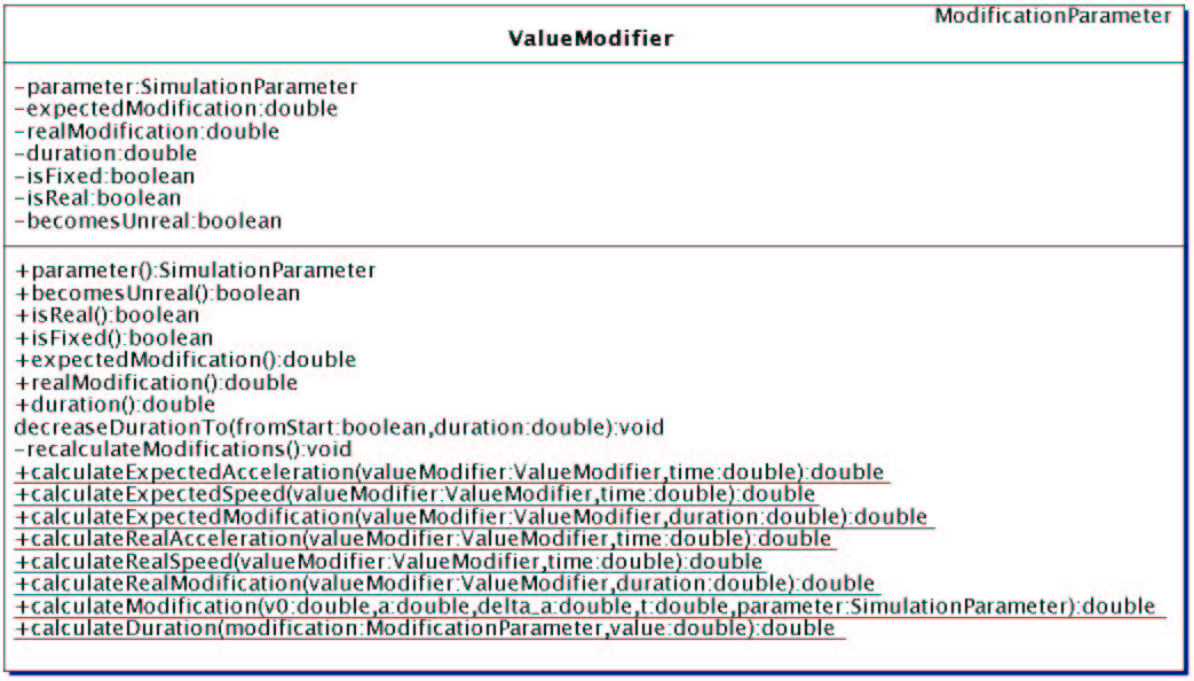


Abbildung 4.7: Bewegungsereignis

die Dauer des Ereignisses entsprechend zu kürzen. Nach einer so erfolgten Kürzung der Ereignisdauer werden die entsprechenden, durch das Ereignis hervorgerufenen, Wertänderungen neu berechnet.

Für die Berechnung dieser Wertänderung sind drei Attribute von besonderer Bedeutung. Sie spiegeln die in Kapitel 3 genannten Arten von Bewegungsereignissen wieder. Zum einen das *„isFixed“* Attribut, welches besagt, dass dieses Ereignis von dem Schrittmotor erzeugt wurde und somit nicht von nachfolgenden Ereignissen gelöscht werden darf. Ist dieses Attribut nicht gesetzt, handelt es sich um eine extern erzeugte Bewegung, die von Motorbewegungen überschrieben wird. Das zweite Attribut *„isReal“* gibt Auskunft darüber, ob das Ereignis Auswirkung auf die reelle Motorstellung hat. Dies wird benötigt, um ein entsprechendes Verhalten bei Überschreitung eines vorgegebenen Drehmomentes zu erlauben. Das *„becomesUnreal“* Attribut spezifiziert, ob ein Ereignis während seines zeitlichen Verlaufs seinen Zustand dahingehend verändert, dass die Bewegung nicht mehr Einfluss auf die reelle Motorbewegung hat.

Die beschriebenen Bewegungsereignisse werden für die Simulation in einem Vektor verwaltet. Dabei werden neue Ereignisse immer an das Ende angefügt und die Abgearbeiteten am Anfang gelöscht. Für die Verwaltung der Ereignisse sowie die Realisierung der Simulation sind die in Abb. 4.8 gezeigten Klassen verantwortlich. Die wichtigste Klasse stellt hierbei die *„ValueSimulation“* Klasse dar. Sie beinhaltet neben dem Vektor der Bewegungsereignisse die Grenzwerte der Simulation sowie deren notwendige Parameter. Die Klasse der *„ValueModifier“* wurde bereits in Abb. 4.7 beschrieben und die *„ModificationParameter“* Klasse ist in Abb. 4.5 behandelt worden. Die durch die *„Limits“* Klasse vorgegebene obere und untere Schranke der Bewegungssimulation werden ebenfalls in Abb. 4.5 beschrieben.

Durch die Klasse der *„SimulationParameter“* werden die in Kapitel 3 genannten und für die Simulation benötigten Parameter verwaltet. Dazu gehört die maximal mögliche Beschleunigung a_{sim} sowie die Geschwindigkeit v_{sim}, ab der die Geschwindigkeitskurve mittels einer Zufallsfunktion moduliert wird. Die in Abb. 4.6 gezeigte Klasse *„Distribution“* stellt diese Zufallsfunktion zur Verfügung. Die in der gleichen Abbildung gezeigte *„TimeGenerator“* Klasse wird ebenfalls von den Simulationsparametern benutzt, um der Simulation die Abfrage der ak-

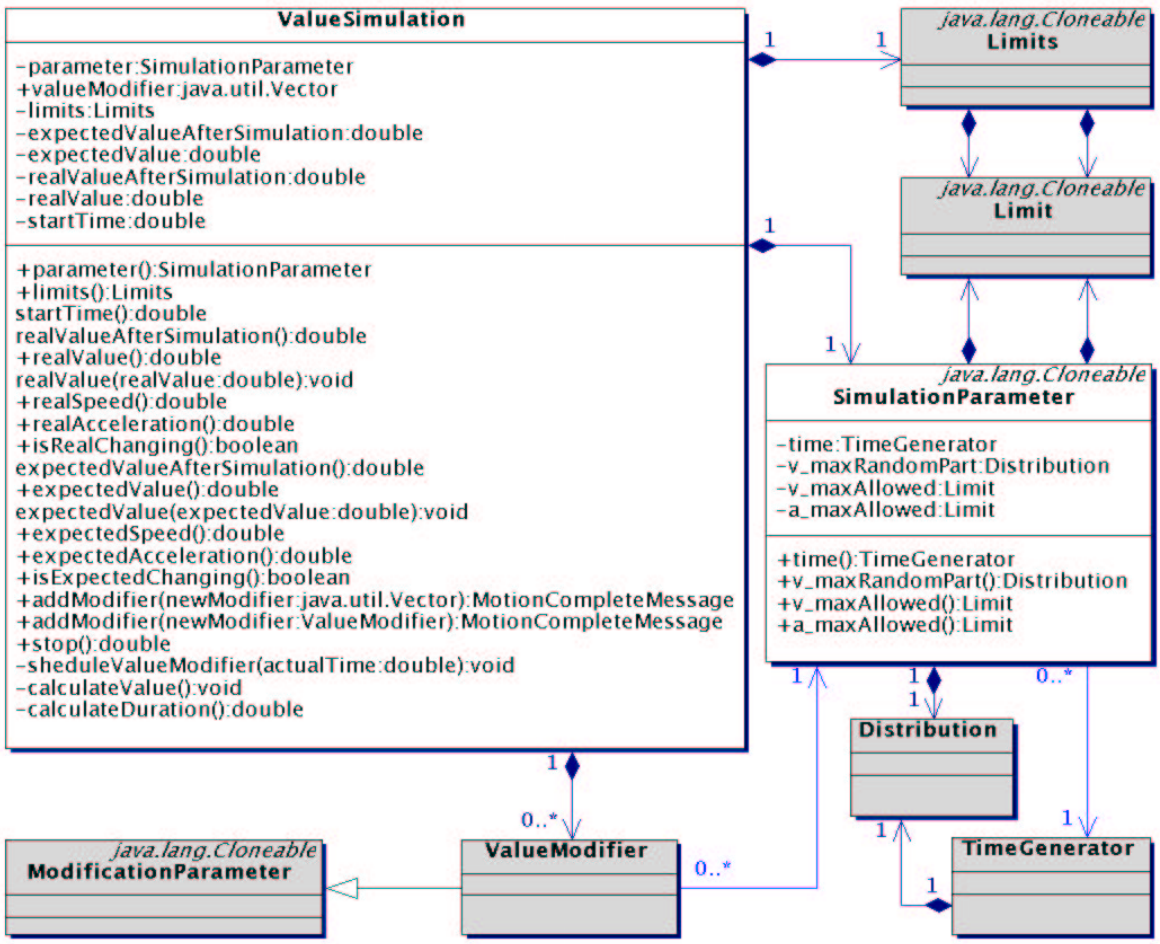


Abbildung 4.8: Klassendiagramm der Simulation

tuellen Zeit zu ermöglichen. In der „*ValueSimulation*“ Klasse wird das in Abb. 3.2 vorgestellte Simulationsschema umgesetzt. In speziellen Attributen werden sowohl der Wert der letzten Berechnung, als auch der Wert, der nach Abarbeitung aller Ereignisse erreicht wird, festgehalten. Dies gilt sowohl für die von der Steuerelektronik erwarteten Werte als auch für die Reellen. Zusätzlich hält ein Objekt der „*ValueSimulation*“ Klasse den Zeitpunkt fest, zu dem die letzte Berechnung vorgenommen wurde. Dieser Zeitpunkt ist identisch mit dem Anfang des ersten Ereignisses des Vektors.

Neben den zahlreichen Methoden zur Bestimmung von Werten beinhaltet die „*ValueSimulation*“ Klasse Methoden, mit denen neue Ereignisse eingefügt werden können und die Simulation gestoppt werden kann. Bei dem Einfügen neuer Ereignisse werden gemäß den Vorüberlegungen Nachrichten erzeugt. Diese Nachrichten werden durch Objekte der „*MotionCompleteMessage*“ Klasse implementiert. In ihnen wird festgehalten, ob Endschalter erreicht werden und nach welcher Zeitspanne das entsprechende Ereignis abgearbeitet ist. Diese Nachrichten werden für die Erzeugung der entsprechenden Antworten der nachgebildeten Steuerelektronik benötigt.

Im nächsten Unterkapitel wird diese Umsetzung der Emulation der Elektronik erläutert und somit auch die Kommunikation zwischen Simulation und nachgebildeter Elektronik.

4.4 Emulation der Elektronik

An die Emulation der Elektronik wurden in Kapitel 3 zwei Hauptanforderungen gestellt. Zum einen ist die nachgebildete Steuerelektronik dafür zuständig, dass die Simulation mit den richti-

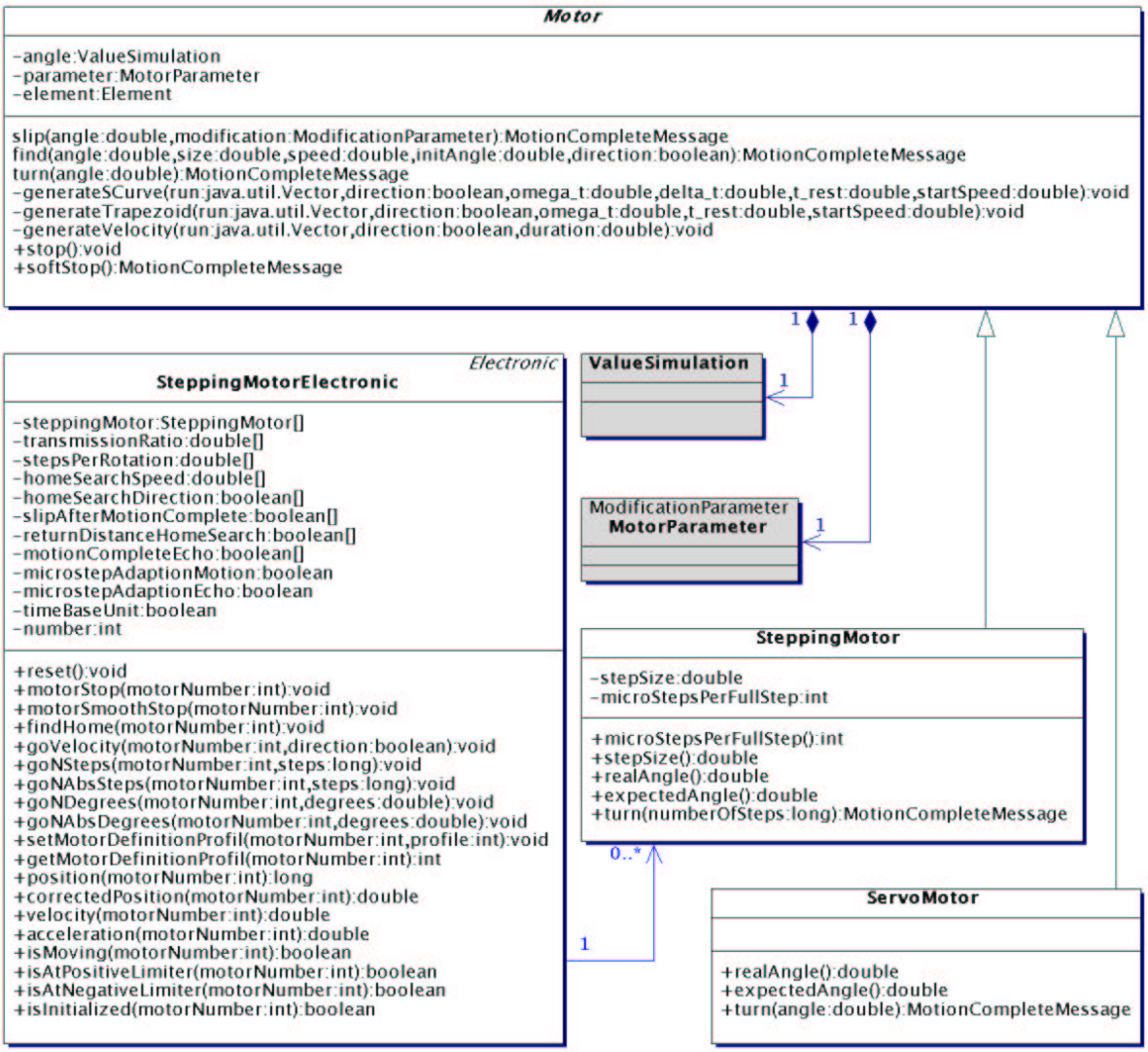


Abbildung 4.9: Schnittstelle zwischen Simulation und Elektronik

gen Ereignissen versorgt wird, zum anderen muss die Kommunikation mit dem Steuercomputer ermöglicht werden. Somit ist dieser Teil zuständig für die Kommunikation zwischen Steuercomputer und Simulation, die über eine Netzwerkverbindung stattfindet. Daher sind die für die Kommunikation notwendigen Klassen in dem separaten *„communication“* Paket realisiert worden.

Da die für die Erzeugung der Ereignisse sowie der Generierung der Nachrichten zuständigen Klassen sehr simulationsspezifisch sind, befinden sich diese ebenfalls in dem *„instrument“* Paket. Zunächst werden die zur Erzeugung der Ereignisse verwendeten Klassen beschrieben.

4.4.1 Ansteuerung der Simulation

Die Ansteuerung der Simulation erfolgt durch die in Abb. 4.9 gezeigten Klassen. Die *„Motor“* Klasse bietet dabei neben den nicht dargestellten Methoden zur Abfrage simulierter Werte, die Möglichkeit, die einem Fahrbefehl entsprechenden Ereignisse zu erzeugen. Diese Ereignisse werden an die entsprechende *„ValueSimulation“* Klasse, die bereits in Abb. 4.8 beschrieben wurde, weitergeleitet. Die für die Erstellung der Ereignisse notwendigen Parameter werden durch die *„MotorParameter“* Klasse repräsentiert (vergleiche Abb. 4.5).Die Ansteuerung der

Simulation erfolgt durch folgende Methoden:

- *„turn“*
  Die *„turn“* Methode erzeugt unter Verwendung der entsprechenden Parameter die für die Drehung um einen angegebenen Winkel notwendigen Bewegungsereignisse. Dies geschieht durch Benutzung der *„generateSCurve“*, der *„generateTrapezoid“* und der *„generateVelocity“* Methode, die die notwendigen Zeiten aus den Formeln 3.11, 3.12 und 3.13 erhalten.

- *„slip“*
  Mit der *„slip“* Methode wird ein Rutschvorgang initialisiert. Dieser Rutschvorgang findet unter Verwendung der mit übergebenen Parameter zu dem ebenfalls angegebenen Motorwinkel statt. Ist dieser Winkel nicht erreichbar, wie z.B. durch schlecht gewählte Parameter, dann wird die Bewegung - soweit es die Parameter zulassen - in Richtung des angegebenen Motorwinkels durchgeführt. Mittels der *„slip“* Methode werden die einen Rastvorgang simulierenden Ereignisse erzeugt.

- *„find“*
  Durch die *„find“* Methode werden die Ereignisse einer Kalibrierungsfahrt erzeugt. Dies geschieht ähnlich der *„turn“* Methode.

- *„stop“*
  Der *„stop“* Befehl bewirkt, dass die Simulation angehalten wird. Die dafür notwendigen Schritte wurden bereits in Kapitel 3 beschrieben.

- *„softStop“*
  Durch die *„softStop“* Methode wir zunächst die Methode zur Beendigung der Simulation aufgerufen. Danach wird entsprechend den vorgegebenen Motorparametern eine langsame Abbremsphase in Form von Ereignissen erzeugt und an die Simulation übergeben.

Da die „Motor“ Klasse sehr allgemein gehalten ist und für jede Art von Motor benutzt werden kann, sind zwei spezialisierte Klassen zu betrachten. Diese Klassen dienen dazu, einen Schrittmotor bzw. Servomotor zu implementieren. Der Servomotor wurde bisher nur für Erweiterungsmöglichkeiten des virtuellen astronomischen Instruments vorgesehen und nicht weiter benutzt. Der Schrittmotor hingegen, ist aufgrund seiner Verwendung im virtuellen astronomischen Instrument, von großer Bedeutung. Um einen Schrittmotor nachbilden zu können, sind seine Schrittgröße sowie die verwendete Mikroschrittunterteilung wichtig. Mit Hilfe dieser Parameter kann ein Schrittmotor einen Fahrbefehl, der in Schritten angegeben wird, auf den in Grad angegebenen *„turn“* Befehl der beerbten *„Motor“* Klasse umsetzen.

Die Ansteuerung der *„SteppingMotor“* Objekte erfolgt über die sehr umfangreiche *„SteppingMotorElectronic“* Klasse. Diese Klasse steuert eine beliebige Menge an Schrittmotoren und bildet somit die Verwaltungsstruktur der reellen Elektronik nach. Außerdem werden in ihr alle in Unterkapitel 3.2 beschriebenen Parameter verwaltet. Diese - sowie die Motorparameter - können durch eine Vielzahl an Methoden verändert bzw. ausgegeben werden. Als Beispiel sind in Abb. 4.9 die *„setMotorDefinitionProfil“* und die *„getMotorDefinitionProfil“* Methode zu erkennen. Alle anderen Methoden dieser Art werden nicht dargestellt, um das Klassendiagramm übersichtlich zu halten. Zusätzlich enthält die *„SteppingMotorElectronic“* Klasse noch Methoden, um die entsprechenden und von der Elektronik erzeugten Bewegungen der Schrittmotoren zu veranlassen. In einem extra Attribut wird außerdem die Nummer der Steuerelektronik festgehalten. Diese wird dazu benutzt, jedes Objekt eindeutig zu bezeichnen.

Die bei dem Erzeugen der Simulationsereignisse generierten Nachrichten werden, wie in Abb. 4.10 gezeigt, dazu benutzt, die Simulation zu steuern bzw. die richtigen Ausgaben zu

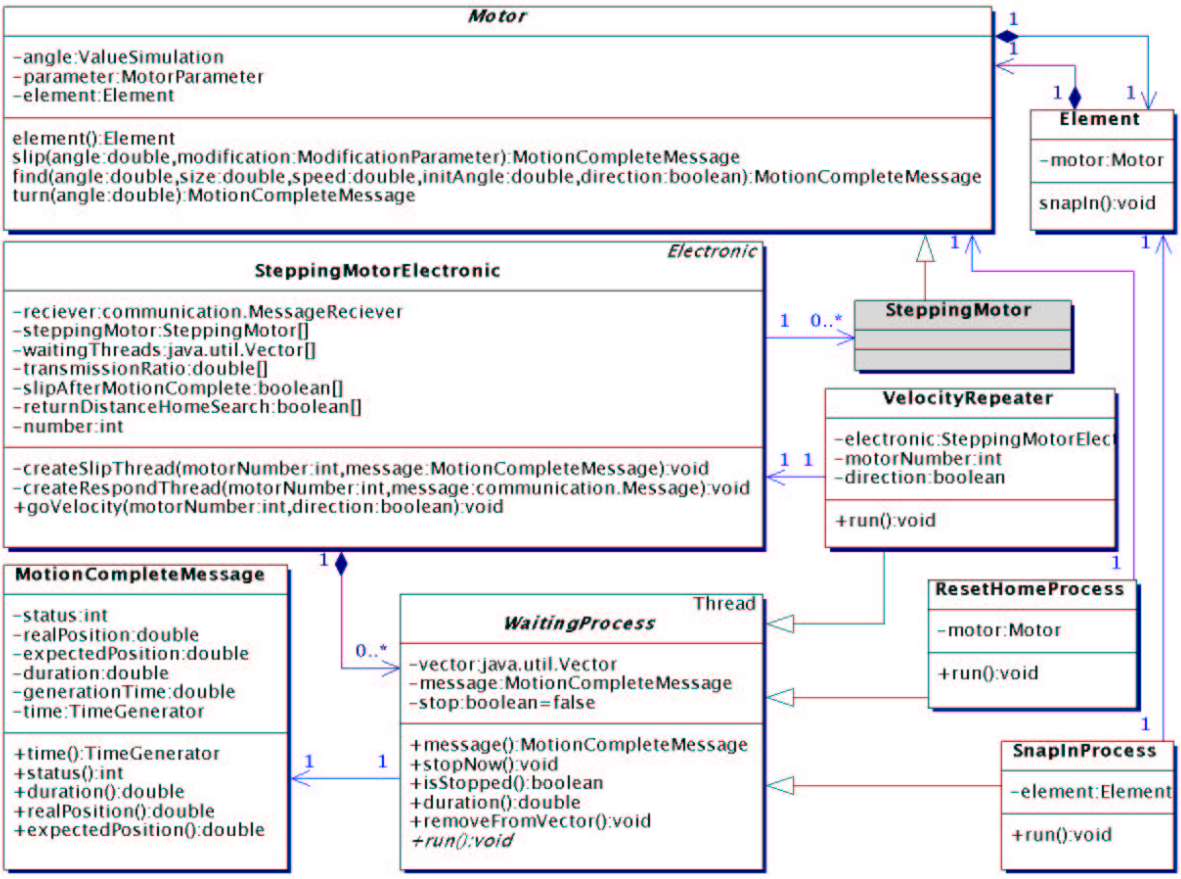


Abbildung 4.10: Verwendung der Nachrichten

erzeugen. Die Objekte der *„SteppingMotorElectronic“* Klasse erhalten nach Aufruf der simulationsbeinflussenden Methoden der *„Motor“* Klasse Nachrichten in Form von *„MotionCompleteMessage“* Objekten. Diese Nachrichten enthalten den Zeitpunkt, nach dem eine Bewegung abgeschlossen ist. Da dieser von dem verwendeten Zeitgenerator der Simulation abhängig ist, besitzt jede Nachricht einen Verweis auf einen Zeitgenerator. Außerdem wird in den *„MotionCompleteMessage“* Objekten gespeichert, ob gegebenenfalls ein Endschalter bzw. der Kalibrierungschalter erreicht wurde. Zusätzlich werden noch die Positionen nach Beendigung des Fahrbefehls festgehalten.

Nachdem ein Objekt der *„SteppingMotorElectronic“* Klasse eine solche Nachricht erhalten hat, wird diese dem verursachenden Befehl entsprechend weiterbearbeitet. Handelt es sich um einen Fahrbefehl, so wird die entsprechende Meldung an die Kommunikationsschnittstelle weitergeleitet. Dieser Prozess wird in Unterkapitel 4.4.3 näher beschrieben. Nach dem Absetzen eines Fahrbefehls wird ein spezieller Thread gestartet, der nach Beendigung der Bewegung die *„snapIn“* Methode des bewegten Elementes aufruft. Dies führt dazu, dass je nach Parametereinstellung der Motor aufgrund eines Einrastvorganges bewegt wird. Für den Fall, dass die Bewegung einen Kalibrierungsvorgang darstellt, wird ebenfalls ein Thread schlafengelegt, der nach erfolgreicher Kalibrierung die Motorpositionen der Simulation synchronisiert. Wenn der Motor sich kontinuierlich und ohne Begrenzung drehen soll, wird auf gleiche Weise durch einen schlafenden Thread die Erzeugung von Bewegungereignissen endlos fortgesetzt. Um sowohl die *„SnapInProcess“* Klasse, die *„ResetHomeProcess“* Klasse als auch die *„VelocityRepeater“* Klasse zu verstehen, die alle für die zuletzt genannten Vorgänge zuständig sind, muss die *„WaitingProcess“* Klasse näher betrachtet werden. Die *„WaitingProcess“* Klasse stellt Möglichkeiten zur verzögerten Handlung bereit. Alle auf das Erreichen eines bestimmten Zeitpunk-

tes wartenden Threads werden von der „*SteppingMotorElectronic*“ Klasse verwaltet. Dies ist dazu nötig, um gegebenenfalls wartende Threads zu beenden. Dazu haben „*WaitingProcess*“ Objekte Methoden, mit denen sie sich beenden lassen oder aus dem verwaltenden Vektor zu löschen sind.

Nachdem alle zur Ansteuerung der Simulation verwendeten Klassen beschrieben wurden, wird auf die Netzwerkanbindung des virtuellen astronomischen Instruments eingegangen.

4.4.2 Netzwerkanbindung

Die „*Network*“ Klasse stellt die Netzwerkanbindung des virtuellen astronomischen Instruments dar. Sie stellt Methoden zum Empfangen und Senden von Nachrichten zur Verfügung. Außerdem werden von ihr die für eine Verbindung notwendigen Daten gespeichert. Um die Nutzungschnittstelle dieser Klasse möglichst kompakt zu halten, implementiert sie die „*MessageReceiver*“ Schnittstelle. Über diese Schnittstelle empfängt ein Objekt der „*Network*“ Klasse die über das Netzwerk zu versendenden Daten (siehe Abb. 4.11).

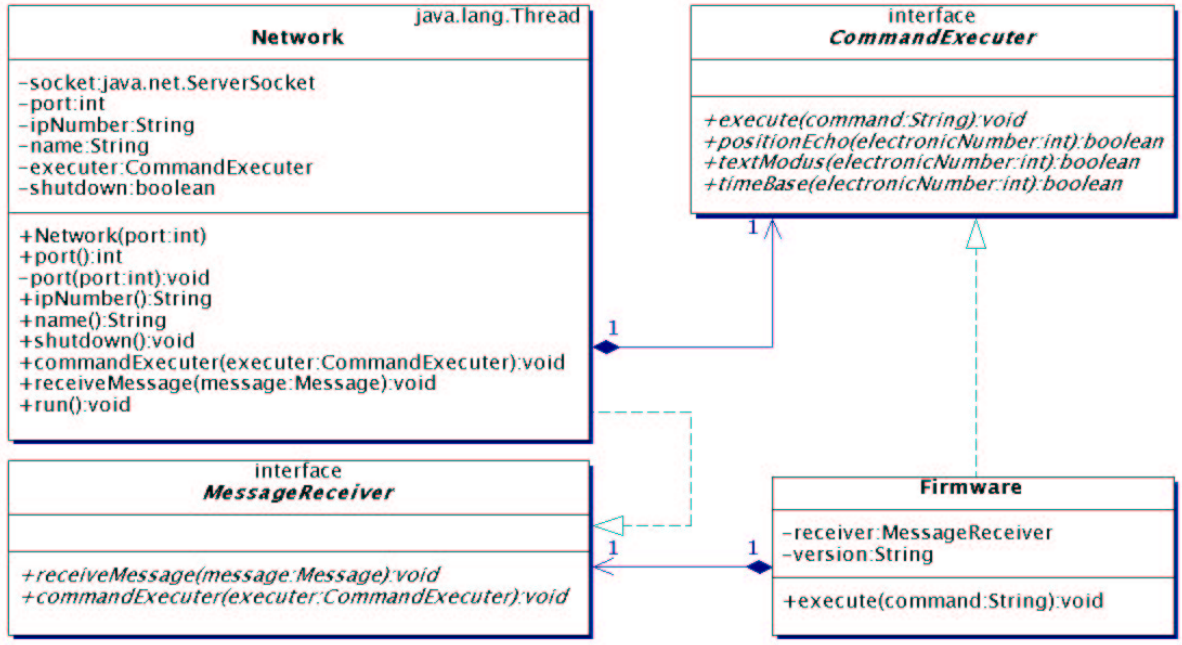


Abbildung 4.11: Netzwerkanbindung

Über den gleichen Mechanismus, wie die „*Network*“ Klasse Daten für den Versand empfängt, werden die von der Netzwerkschnittstelle empfangenen Daten an die „*Firmware*“ Objekte weitergeleitet. Diese implementieren die „*CommandExecuter*“ Schnittstelle und ermöglichen damit anderen Objekten Kommandotexte an sie weiterzureichen. Neben der in Richtung des virtuellen astronomischen Instruments gerichteten Anfragen und den direkt darauf folgenden Antworten, müssen auch zeitversetzte Nachrichten ausgegeben werden.

4.4.3 Ausgabe zeitversetzter Nachrichten

Um Nachrichten, die zeitversetzt über das Netzwerk ausgegeben werden sollen, zu unterstützen, wird die in Abb. 4.10 dargestellte „*WaitingProcess*“ Klasse benutzt. In Abb. 4.12 ist zu erkennen, wie die „*MessageResponder*“ Klasse die „*WaitingProcess*“ Klasse beerbt und somit ihre Eigenschaft der verzögerten Ausführung von Aktionen übernimmt. Die von der Simulation erzeugten „*MotionCompleteMessage*“ Objekte (vergleiche Abb. 4.10) werden von der

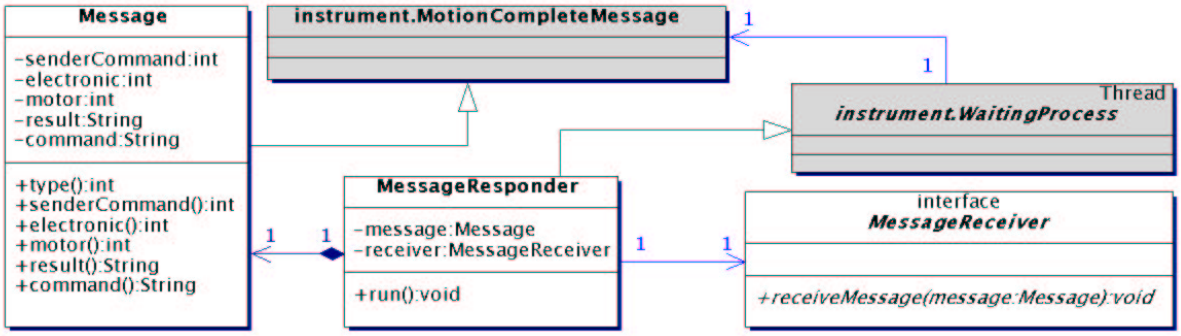


Abbildung 4.12: Nachrichtenausgabe über die Netzwerkverbindung

entsprechenden „*Message*“ Klasse des „*communication*“ Paketes spezialisiert. Sie werden um das Kommando, das diese Antwort verursacht hat, erweitert. Außerdem ist in ihr alles übrige enthalten, was von der „*Network*“ Klasse benötigt wird, um die entsprechenden Daten zu formatieren und über das Netzwerk zu schicken. Nach Ablauf der in der Nachricht festgehaltenen Zeit, wird diese in einem „*MessageResponder*“ Objekt aufbewahrte Nachricht über die „*MessageReceiver*“ Schnittstelle an das entsprechende „*Network*“ Objekt weitergeleitet. Dieses formatiert die Nachricht anhand der in ihr enthaltenen Daten und verschickt diese über die Netzwerkschnittstelle an den Steuercomputer.

Somit ist die Richtung der Daten von dem virtuellen astronomischen Instrument hin zu dem Steuercomputer beschrieben. Als nächstes wird erläutert, wie die Daten empfangen, ihre Bedeutung entschlüsselt und die entsprechenden Befehle ausgeführt werden.

4.4.4 Ausführen der Kommandos

Eine wichtige Anforderung an das virtuelle astronomische Instrument ist, dass es in der Lage sein soll, die gleichen Steuerkommandos wie die reale Elektronik zu empfangen und genauso wie diese zu reagieren. Dafür wurde ein Simulationsmodell entworfen und implementiert. Außerdem wurden Ansteuermöglichkeiten für die Simulation geschaffen. Als letzte Aufgabe bleibt die richtige Umsetzung der empfangenen Steuerkommandos auf die entsprechenden Methoden, die die Simulation beeinflussen, zu betrachten. Damit ein gleiches Verhalten wie bei der reellen Steuerelektronik nachgebildet werden kann, das zudem noch sehr flexibel ist, ist es notwendig die Firmware der Elektronik nachzubilden.

Nachbildung Firmware

Die Firmware bezeichnet das Programm, welches das Verhalten der in der reellen Elektronik verwandten Bausteine steuert. Aufgabe der Firmware ist es, die Steuerkommandos zu erkennen und entsprechend zu handeln. Genauso wie bei der reellen Firmware muss auch ihr virtuelles Gegenstück dieser Aufgabe nachkommen. Um dies realisieren zu können, verwendet die Firmware die in Abb. 4.13 gezeigten Klassen. In der „*Firmware*“ Klasse selbst werden ein paar zusätzliche Parameter verwaltet, die ihr Verhalten beeinflussen. Kommt über die in Abb. 4.11 gezeigte Schnittstelle ein Kommando an, so wird dieses zuerst mit Hilfe der „*CommandParser*“ Klasse zerlegt. Danach wird das Kommando anhand einer Tabelle mit „*Command*“ Objekten identifiziert und die dementsprechenden Methoden der jeweiligen nachgebildeten Steuerelektronik aufgerufen. Kann ein Kommando nicht einem Eintrag aus den Kommandotabellen zugeordnet werden, so werden die entsprechenden Fehlermeldungen erzeugt. Dies geschieht analog zur Erzeugung von Fehlermeldungen der reellen Firmware.

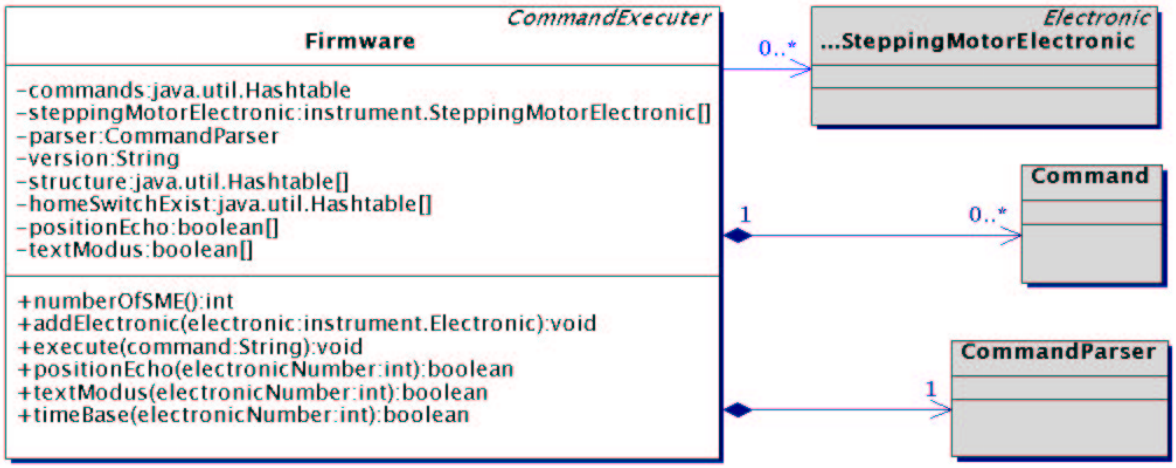


Abbildung 4.13: Nachbildung der Firmware

Als nächstes wird die Datenstruktur, mit Hilfe derer die Kommandos sehr flexibel definiert werden können, beschrieben.

Datenstruktur der Kommandos

Um die stark im Wandel befindlichen Kommandos der Steuerelektronik nachbilden zu können, wurde in Kapitel 3 eine baumartige Struktur vorgestellt. Diese Struktur zur Kommandodefinition findet sich in Abb. 4.14 wieder. Die zentrale *„Command“* Klasse beinhaltet neben dem Kommando einen kurzen beschreibenden Text. Dieser wird später in der Visualisierung der Kommunikationsdaten (siehe Abb. 4.16) verwendet, um die Bedeutung eines Kommandos zu beschreiben. Das Kommando selbst wird aus einer symbolisierenden Zeichenkette und einer Typenkennung zusammengesetzt. Anhand der Zeichenkette wird innerhalb eines Kommandos dessen Typ erkannt und somit festgelegt, wie die Firmware das Kommando zu bearbeiten hat. Mittels einer alle Kommandos umfassenden Tabelle ist es möglich, diese entsprechend ihres Typs zu entschlüsseln.

Jede Kommandodefinition setzt sich aus der entsprechenden Zeichenkette und einer Menge möglicher Parameter zusammen. Diese Parameter werden durch die *„CommandParameter“* Klasse beschrieben. Auf jeden Parameter kann entweder ein weiterer Parameter folgen oder es ist eine Auswahl an Parametern möglich. Diese Auswahl geschieht durch die *„Selektion“* Klasse, in der mittels einer Tabelle auf andere *„CommandParameter“* Objekte verwiesen wird. Die *„CommandParameter“* Objekte verfügen über eine Klassifizierung sowie über eine eventuelle Wertbeschränkung. Damit kann vorgegeben werden, welche Form ein Parameter hat. Zur Auswahl stehen beschränkte und unbeschränkte reelle Zahlen, beschränkte und unbeschränkte ganze Zahlen, beschränkte und unbeschränkte Hexadezimalwerte sowie Binärwerte. Außerdem verfügen alle Parameter über eine Positionsangabe bezüglich ihres Vorkommens in einer Kommandozeile. Anhand dieser Kommandodefinition kann die Firmware dann die Korrektheit der entsprechenden Eingaben überprüfen und auswerten.

Zuständig für das richtige Zerlegen der ankommenden Nachrichten ist der Kommandoparser.

Kommandoparser

Der Kommandoparser zerteilt anhand der ihm zur Verfügung stehenden Daten, eine Kommandonachricht in die Nummer der Steuerelektronik, das Kommando selbst und die entsprechenden

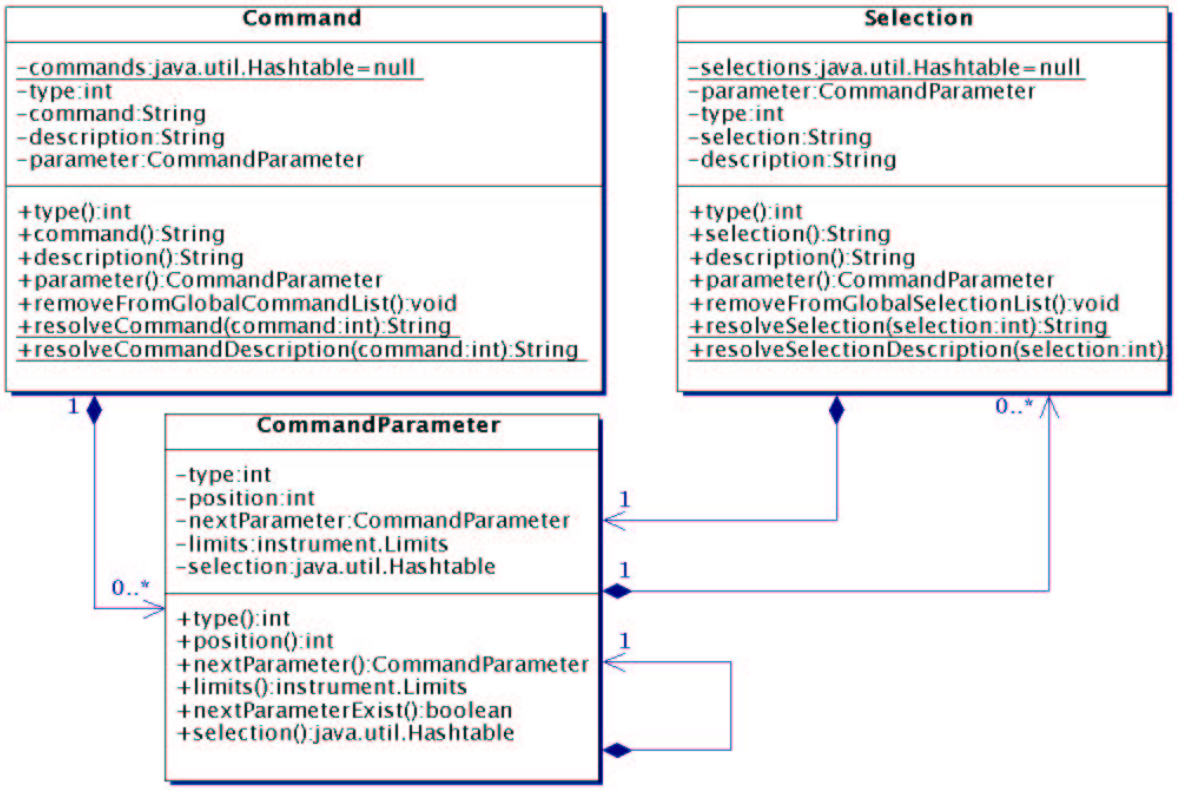


Abbildung 4.14: Kommandostruktur

Kommandoparameter. Die Daten über die Position der Elektroniknummer sowie des Kommandos entnimmt er der in Abb. 4.15 gezeigten *„CommandPositions“* Klasse. Anhand einer der *„CommandParser“* Klasse bekannten Liste von Trennzeichen, Elektroniknummern und Kommandowörten wird eine übergebene Textnachricht zerteilt und eingelesen. Dabei wird das Ergebnis durch ein Objekt der *„ParseResult“* Klasse repräsentiert. In ihm werden die gelesenen Elektroniknummer sowie das entsprechend erkannte Kommando festgehalten. Außerdem enthält es die separierten Parameter. Mit diesen Ergebnissen kann die Firmware das angegebene Kommando mit den entsprechenden Parametern von der richtigen, nachgebildeten Elektronik ausführen lassen.

Treten Fehler während der Kommandoerkennung auf, wird eine entsprechende Ausnahmebehandlung aktiviert. Diese findet in Form der *„ParseException“* Klasse statt und muss von der Firmware entsprechend behandelt werden. Während der Kommandoerkennung werden nur Fehler bei der Elektroniknummer, den Kommandozeichen sowie falscher Parametertrennung berücksichtigt. Ob Parameter nicht mit der Kommandodefinition übereinstimmen wird in der Firmware überprüft.

Das nächste und somit auch letzte der drei Unterkapitel beschäftigt sich mit der Visualisierung des virtuellen Instruments.

4.5 Visualisierung

Wie bei den Vorüberlegungen bereits beschrieben wurde, müssen zwei Bereiche von Daten visualisiert werden. Zum einen ist die Visualisierung der Kommunikationsdaten zu nennen, zum anderen die der Parameter der mechanischen Bestandteile des Instruments. Im Gegensatz zu dem Softwarepaket für die Simulation der Mechanik und dem für die Emulation der Elektronik ist das Softwarepaket, welches für die Visualisierung zuständig ist, nicht eigenständig zu

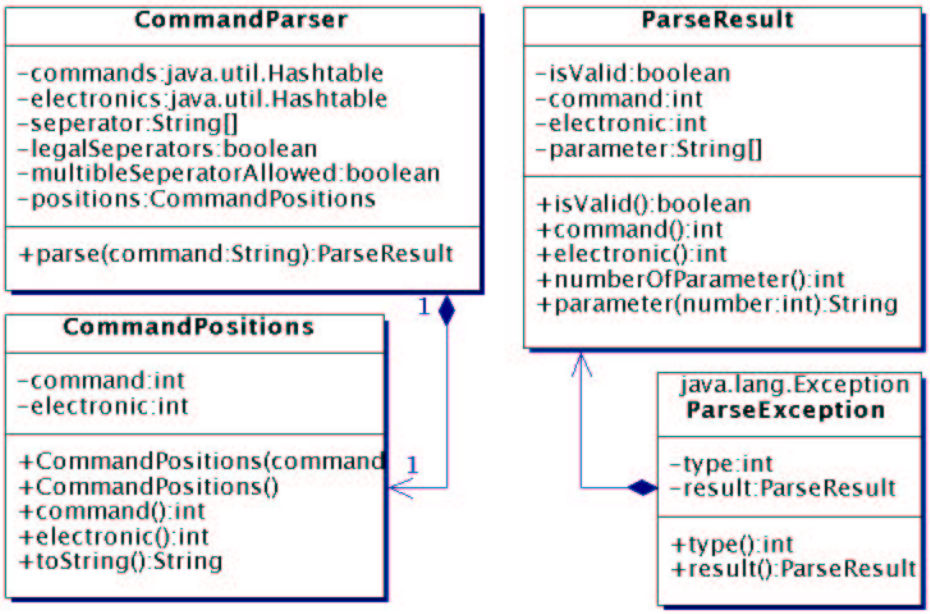


Abbildung 4.15: Kommandoparser

benutzen. Es benötigt die anderen beiden Pakete, da die Visualisierungen speziell auf diese zugeschnitten sind.

Für die Implementierung der graphischen Benutzungsoberfläche wurde die „*Swing*“ Bibliothek von Java verwendet. Diese Wahl beeinflusst den objektorientierten Modellierungsansatz, was sich in den später folgenden Klassendiagrammen widerspiegelt. Nähere Informationen über die Swing Bibliothek befinden sich unter [LW98]. Mit Hilfe der Informationen dieses Buches ist es leicht möglich, eine ansprechende graphische Benutzungsoberfläche zu schaffen.

Da die Visualisierung streng den entsprechenden Aufgaben nach unterteilt worden ist, kann diese auch den Aufgabengebieten nach getrennt beschrieben werden. Als erstes wird hierbei auf die Visualisierung der Kommunikationsdaten eingegangen.

4.5.1 Visualisierung der Kommunikationsdaten

Bei der Visualisierung der Kommunikationsdaten sollen sämtliche von dem virtuellen astronomischen Instrument empfangenen und gesendeten Daten dargestellt werden. Dies soll in Form einer Liste geschehen, die nach dem zeitlichen Auftreten der Daten sortiert ist. Die Visualisierung dieser Daten wird von einer einzigen Klasse übernommen. Die Interaktion der „*NetworkDataObserver*“ Klasse mit den entsprechenden Klassen des „*communication*“ Pakets geht aus Abb. 4.16 hervor. Die einzelnen Klassen in dem gezeigten UML-Diagramm wurden auf die wichtigsten Attribute und Methoden reduziert, um eine bessere Übersicht zu ermöglichen.

Um eine Schnittstelle zwischen dem „*communication*“ und dem „*visualisation*“ Paket zu schaffen, ist in dem „*communication*“ Paket das „*ProtocollReceiver*“ Interface definiert worden. In diesem Interface wird festgelegt, dass Klassen, die Protokolldaten empfangen sollen, eine „*receiveProtocollData*“ Methode implementieren müssen. Diese einfache Methode benutzt auch die „*NetworkDataObserver*“ Klasse, um über gesendete oder empfangene Daten informiert zu werden.

Mit dem Aufruf der „*receiveProtocollData*“ Methode wird ein Objekt vom Typ „*ProtocollData*“ übergeben. Diese Objekte enthalten die jeweiligen Kommunikationsdaten, einen kurzen beschreibenden Text, die Art der Meldung (ok, Warnung, Fehler), ob die Daten gesendet oder empfangen wurden und den entsprechenden Zeitpunkt. Nach Empfang dieses Objekts wird es

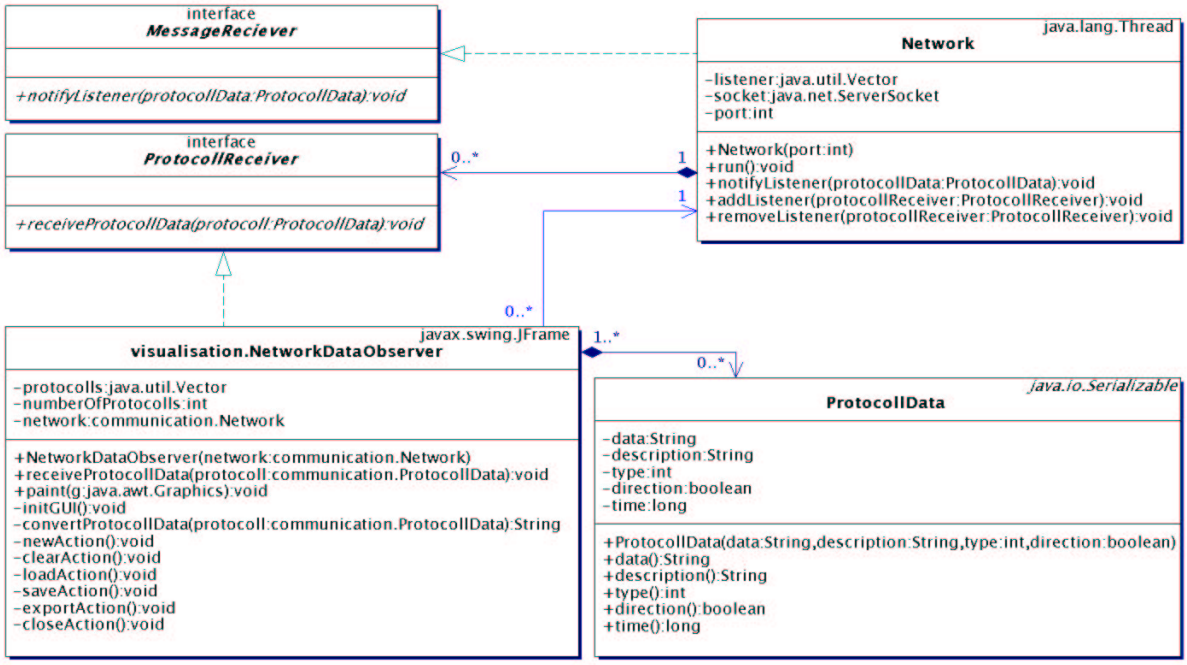


Abbildung 4.16: Visualisierung der Kommunikationsdaten

als erstes in einem Vektor gespeichert, direkt danach den gewünschten Selektionen entsprechend textuell aufbereitet und anschließend dargestellt. Diese Darstellung erfolgt in Form eines formatierten HTML-Textes innerhalb eines entsprechenden Elements der „*Swing*“ Bibliothek.

Die „*NetworkDataObserver*“ Klasse stellt außerdem Methoden zur Verfügung, um ein neues „*NetworkDataObserver*“ Objekt zu erstellen oder ein bereits bestehendes Objekt zu schließen, seinen Inhalt zu löschen, zu speichern, zu laden oder im HTML-Format zu exportieren.

Bei der Initialisierung eines neuen „*NetworkDataObserver*“ Objekts trägt sich dieses in dem entsprechenden zu überwachenden „*Network*“ Objekt in einer Liste ein. Wird ein „*NetworkDataObserver*“ Objekt nicht mehr benötigt, trägt es sich selbstständig wieder aus dieser Liste aus.

Wenn ein „*Network*“ Objekt Daten empfängt, oder über das „*MessageReceiver*“ Interface dazu veranlasst wird, Protokolldaten weiterzuleiten, wird für jedes Objekt, das sich in der bereits genannten Liste eingetragen hat, die „*receiveProtocollData*“ Methode mit den entsprechenden Kommunikationsdaten aufgerufen. Dieses Weiterleiten von Daten geschieht innerhalb der „*notifyListener*“ Methode. Somit empfangen alle eingetragenen Visualisierungskomponenten die gleichen Kommunikationsdaten. Für noch detailliertere Informationen über die Visualisierung der Kommunikationsdaten sei hier auf die sehr umfangreich erstellte Dokumentation des „*visualisation*“ und „*communication*“ Pakets verwiesen. Diese Dokumentation liegt den entsprechenden Softwarepaketen in HTML-Form bei. Eine Beschreibung der graphischen Benutzungsoberfläche der Visualisierung der Kommunikationsdaten befindet sich in Kapitel 5.

4.5.2 Visualisierung der Mechanik

Die Visualisierung der Mechanik sollte sich, wie in den theoretischen Vorüberlegungen bereits besprochen, aus drei unterschiedlichen Ansichten zusammensetzen. Die Strukturansicht findet sich in dem „*ElementTree*“ Objekt der „*MainWindow*“ Klasse wieder. Außerdem ist eine Strukturansicht der Unterelemente eines Elements in jeder Kontextansicht, die durch die „*ElementWindow*“ Klasse definiert wird, vorhanden. Als letzte noch nicht genannte Ansicht bleibt die geometrische zu nennen. Sie wird durch die „*Instrument3d*“ Klasse realisiert. Eine grobe

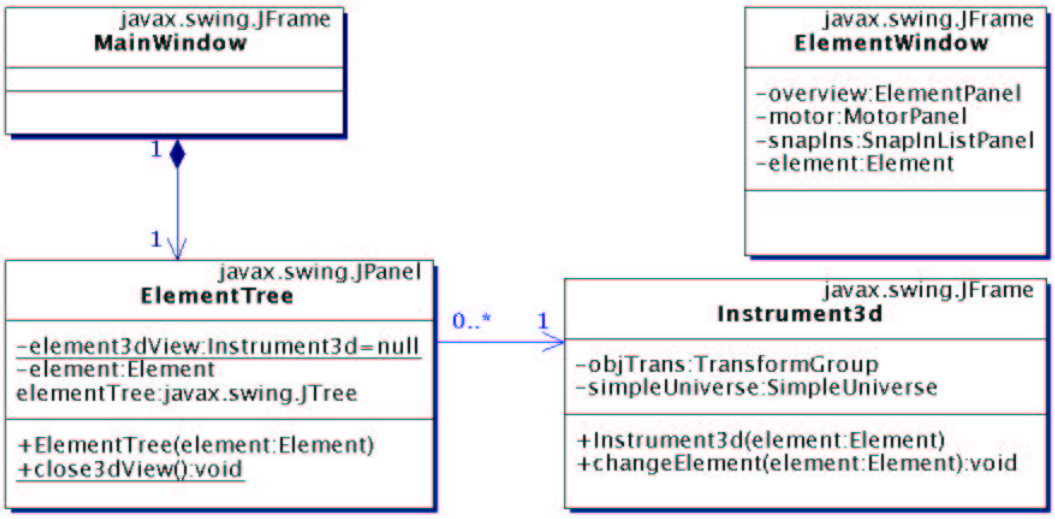


Abbildung 4.17: Visualisierungsansichten der Mechanik

Übersicht vermittelt Abb. 4.17. In diesem Klassendiagramm ist die Dreiteilung besonders gut zu sehen. Es gibt insgesamt drei Fenster, die für die Strukturansicht, Kontextansicht und die geometrische Ansicht verwendet werden. Dies sind die *„MainWindow“*, die *„ElementWindow“* und die *„Instrument3d“* Klasse. Alle drei Klassen sind abgeleitet von der *„JFrame“* Klasse der *„Swing“* Bibliothek und somit eigenständige und vollwertige Fenster. Die Strukturansicht selbst ist kein eigenständiges Fenster, sondern vielmehr eine Möglichkeit zur strukturierten Darstellung des Instrumentes. Sie wird sowohl in dem *„MainWindow“* Objekt eingebunden, um die Gesamtstruktur darzustellen, als auch in dem *„ElementWindow“* genutzt, um Teilstrukturen zu visualisieren.

Natürlich ist jeweils eine Klasse für die Visualisierungsaufgaben nicht mächtig genug; vielmehr findet die Visualisierung durch Interaktion der verschiedensten Klassen statt. Diese Interaktion soll innerhalb der nächsten Seiten erläutert werden.

Bevor sich jedoch direkt der Visualisierung der Mechanik zugewendet wird, ist es nötig, die Datenstruktur, die die mechanischen Elemente speichert, sowie den Modellgraphen, der für die dreidimensionale Darstellung benötigt wird, genauer zu betrachten.

Modellgraph

Da Java3d für die dreidimensionale Visualisierung des virtuellen Instruments verwendet werden soll, werden zunächst ein paar wesentliche und grundlegende Eigenschaften von Java3d behandelt.

Java3d ist eine objektorientierte Bibliothek. Einzelne Anwendungen können individuelle graphische Objekte erzeugen und diese in einer baumähnlichen Struktur, zu sogenannten Szenen, im englischen als *„scene graph“* bezeichnet, zusammenfügen. Diese Szenen können danach von dem Programm durch vorher definierte Schnittstellen verändert werden. Bestandteile dieser Baumstruktur sind ein virtuelles Universum, das die Wurzel des Baumes bildet, Ortsobjekte, sogenannte *„Locale“* Objekte, die die Position einer darunterliegenden Szene innerhalb des Universums beschreiben, sowie Gruppierungs- und Transformationsknoten, durch die Unterbäume zusammengekettet oder verändert werden. Natürlich bietet Java3d noch eine weit größere Zahl an Elementen, die in einer Szene verwendet werden können. Zur Wahrung der Übersichtlichkeit seien jedoch nur die oberen vier genannt. Für nähere Information ist [SRD00] zu empfehlen. Ein einfaches Beispiel für einen *„scene graph“* stellt Abb. 4.18 dar. In dieser Szenenstruktur sieht man, dass in dem virtuellen Universum ein Ort existiert, an dem wieder-

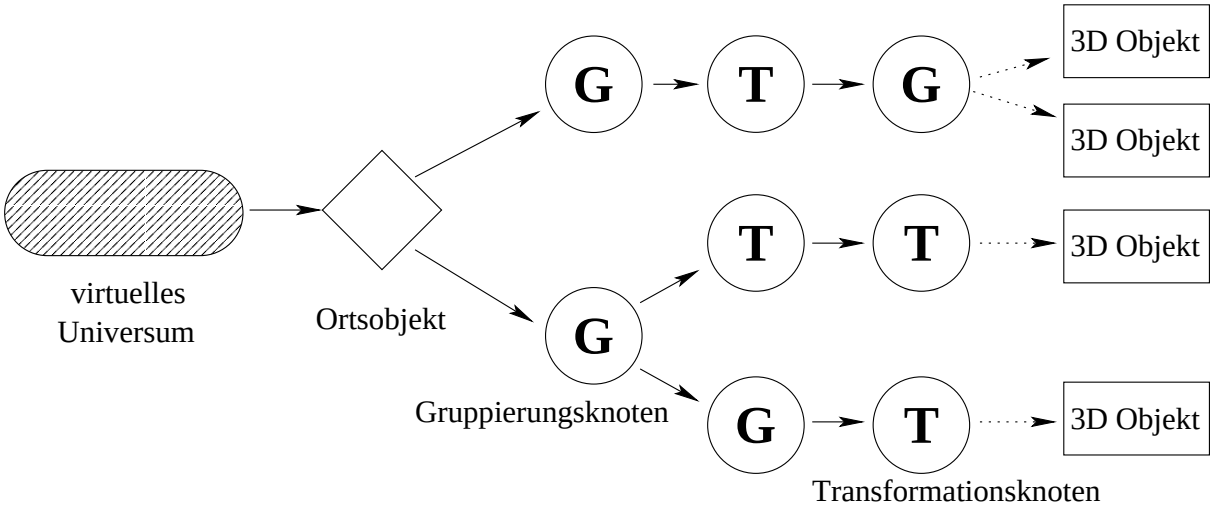


Abbildung 4.18: Einfache Java3d Szenenstruktur

um zwei Unterbäume hängen. Die mit (G) gekennzeichneten Knoten stellen Gruppierungsknoten dar, an denen sich jeweils beliebig viele andere Knoten befinden können. Aus Abb. 4.18 geht außerdem hervor, wie die mit (T) bezeichneten Transformationsknoten eingesetzt werden. Ein Transformationsknoten darf im Gegensatz zum Gruppierungsknoten nur jeweils einen ihm nachfolgenden Knoten besitzen. Im rechten Teil der Abb. 4.18 sieht man die vereinfacht dargestellten dreidimensionalen Objekte, die entsprechend den (T) und (G) Knoten gruppiert und transformiert werden. Wenn auf einem Wege von der Wurzel der Szene (dem virtuellen Universum Objekt) ein Transformationsknoten erreicht wird, wirkt sich dessen Transformation auf alle ihm nachfolgenden Knoten und somit auch Objekte aus.

Der in Java3d verwendete Ansatz des Szenenaufbaus innerhalb einer Baumstruktur ermöglicht es, einfach und flexibel, dreidimensionale Objekte darzustellen, so dass der Programmentwickler sich nicht mit den elementaren Problemen einer 3d Darstellung beschäftigen muss.[2]

Transformationen werden in Java3d durch 4x4 Matrizen gegeben. Dabei wird die Veränderung, die eine Matrix auf einen Vektor oder eine Position bewirkt, wie folgt berechnet:

$$\begin{Bmatrix} m00 & m01 & m02 & m03 \\ m10 & m11 & m12 & m13 \\ m20 & m21 & m22 & m23 \\ m30 & m31 & m32 & m33 \end{Bmatrix} * \begin{Bmatrix} x \\ y \\ z \\ w \end{Bmatrix} = \begin{Bmatrix} x' \\ y' \\ z' \\ w' \end{Bmatrix}$$

$$\Rightarrow \begin{array}{l} x' = m00 * x + m01 * y + m02 * z + m03 * w \\ y' = m10 * x + m11 * y + m12 * z + m13 * w \\ z' = m20 * x + m21 * y + m22 * z + m23 * w \\ w' = m30 * x + m31 * y + m32 * z + m33 * w \end{array} \quad (4.1)$$

Nachdem die Grundlagen der Struktur, die notwendig sind, um dreidimensionale Graphiken mit Java3d zu erzeugen, erläutert wurden, wird die für die Visualisierung notwendige Datenstruktur vorgestellt.

Datenstruktur der Mechanik

Die zur dreidimensionalen Visualisierung der Mechanik notwendige Datenstruktur orientiert sich an der von Java3d vorgegebenen Baumstruktur. Diese Datenstruktur wird für die Strukturansicht und die geometrische Darstellung benutzt. Abb. 4.19 zeigt ein vereinfachtes Klassendiagramm, aus dem die Verkettung der für die geometrische Darstellung wichtigen Klassen sowie

[2]Beispiele hierfür sind: Triangulierung, Renderingalgorithmen oder Interaktion von Objekten.

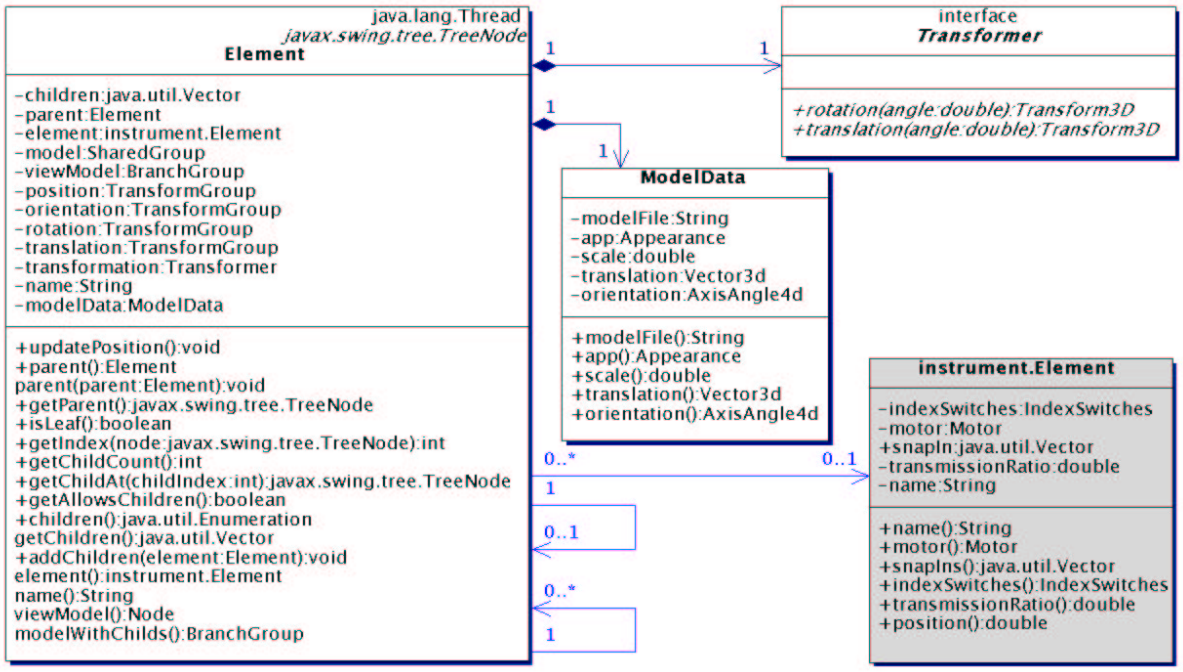


Abbildung 4.19: Datenstruktur der Visualisierung

deren Attribute hervorgeht. In Abb. 4.19 ist außerdem die Verknüpfung der für die Visualisierung notwendigen Klassen mit der Datenstruktur des Instruments zu erkennen.

Die zentrale Klasse der Datenstruktur der Visualisierung ist die „*Element*“ Klasse. Sie bildet das Gegenstück zu einem „*Element*“ aus dem für die Simulation benötigten „*instrument*“ Paket. Ein Attribut der „*Element*“ Klasse verweist deshalb auf das entsprechende „*Element*“ des „*instrument*“ Paketes. Durch diese Verknüpfung ist es möglich, dass mehrere Visualisierungselemente für ein mechanisches Element benutzt werden.

Die genannte baumartige Struktur entsteht dadurch, dass jedes „*Element*“ beliebig viele Kinder, das heißt Objekte die diesem „*Element*“ untergeordnet sind, besitzen kann. Außerdem besitzt jedes Element einen Verweis auf das Objekt seines Vorfahren, dem es untergeordnet ist. Aus Abb. 4.20 geht eine einfache baumartige Verknüpfung zwischen den „*Element*“ Objekten hervor.

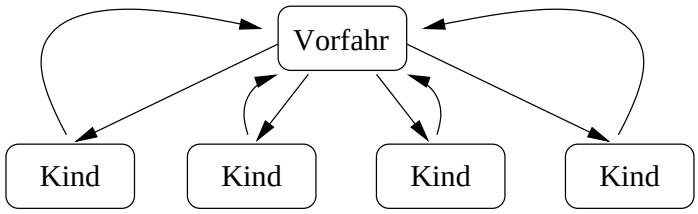


Abbildung 4.20: Baumartige Elementstruktur

Diese Verknüpfung wird in in der „*Element*“ Klasse durch einen Vektor, der alle Kinder enthält, sowie einem Verweis auf das vorangegangene Objekt realisiert. Auf die Kinder sowie den Vorfahren eines „*Element*“ Objekts kann über die beiden „*parent*“ Methoden sowie die „*getChildren*“ und die „*addChildren*“ Methode Einfluss genommen werden.

Neben dieser Verknüpfung der einzelnen „*Element*“ Objekte, die der Struktur eines Szenegraphen ähnelt, besitzt jedes „*Element*“ weitere für die Visualisierung notwendige Attribute. Für die textuelle Darstellung wird der Name jedes Visualisierungsobjektes in einem eigenen Attribut festgehalten. Die geometrische Darstellung erfordert weitere Attribute, die die Position,

Ausrichtung des dreidimensionalen Objekts sowie dessen Ausrichtungs- und Positionsänderung beschreiben. Dies geschieht durch die bereits genannten Transformationsknoten und die in ihnen enthaltenen Transformationsmatrizen.

Die Ausrichtungs- und Positionsänderung geschieht durch die Verwendung der *„Transformer“* Schnittstelle. Jedes Visualisierungselement besitzt ein die *„Transformer“* Schnittstelle implementierendes Objekt. Dieses Objekt erzeugt dem übergebenen Winkel entsprechende Transformationsknoten für die Positions- und Ausrichtungsänderung, die bei jedem Aktualisierungsaufruf die Transformationsknoten des *„Element“* Objekts ersetzen. Dadurch, dass diese *„Transformer“* Objekte jede mögliche Transformationsmatrix in Abhängigkeit des Winkels erzeugen können, wird es mit ihnen möglich, jede Bewegung im dreidimensionalen Raum darzustellen. Damit diese Bewegung mit der aktuellen Position der Simulation synchronisiert wird, stellt jedes *„Element“* Objekt einen eigenen Thread dar, der in einem vorgegebenen Zeitintervall diese *„Transformer“* Objekte aufruft und somit die Ausrichtung und Position aktualisiert.

Als letztes sind noch die Attribute zu nennen, die das dreidimensionale Objekt beschreiben. In dem *„ModelData“* Objekt, das eindeutig einem *„Element“* zugeordnet ist, werden diese Daten festgehalten. Hierzu zählen der Dateiname der 3D-Daten, das Erscheinungsbild (wie z.B. Farbe, Transparenz, Leuchten), ein Skalierungsfaktor sowie die Positions- und Ausrichtungsdaten für den Szenegraphen. Jedes *„Element“* Objekt besitzt jeweils noch ein *„model“* und ein *„viewModel“* Attribut. In dem *„viewModel“* Attribut wird nur das dreidimensionale Modell festgehalten, von dem auf Anfrage Kopien erstellt und ausgegeben werden. Das *„model“* Attribut hingegen enthält die gesamte Szenenstruktur des Elements inklusive aller seiner Kinder, die mittels Verknüpfungen verbunden sind. Dies geschieht in Java3d durch sogenannter *„SharedGroup“* und *„Link Leaf Node“* Objekte. Der Aufbau dieser Szenenstruktur ist Abb. 4.21 zu entnehmen.

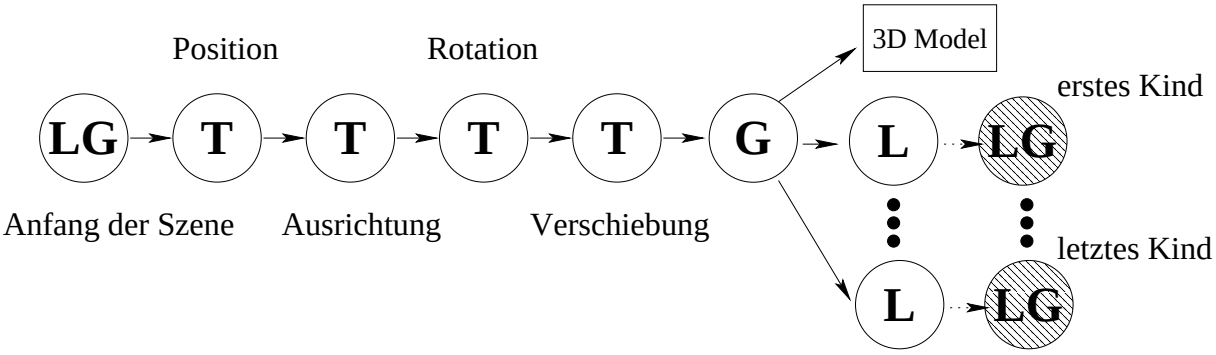


Abbildung 4.21: Benutzte Szenenstruktur

Nach dem verknüpfbaren Gruppierungsknoten (LG) folgt eine Reihe von Transformationsknoten, die zuständig sind für die Position, Ausrichtung, Rotation und Verschiebung des dreidimensionalen Objektes. Die Transformationsknoten, die zuständig sind für die Rotation und Verschiebung des Objektes, werden durch den oben genannten *„Transformer“* entsprechend verändert. Auf diese vier Transformationsknoten folgt dann ein Gruppierungsknoten, mit dem das dreidimensionale Modell des *„Elemente“* Objekts verbunden ist. Außerdem werden die Kinder des Objektes, genauer gesagt deren linkbare Gruppierungsknoten, durch Verknüpfungsknoten *„L“* mit diesem Gruppierungsknoten verbunden.

Zu der Datenstruktur der Mechanik gehören neben den bereits vorgestellten Klassen natürlich auch die Klassen, die notwendig für die Simulation des virtuellen Instruments sind. Diese wurden bereits vorher in Zusammenhang mit der Simulation erwähnt (siehe Abb. 4.5). Diese Datenstruktur wird über die *„Element“* Klasse des *„instrument“* Paketes, wie in Abb. 4.19 gezeigt, eingebunden.

Nachdem die Datenstrukturen erklärt wurden, auf denen die Visualisierung aufbaut, wird nun zur Ersten der drei Ansichten übergegangen; der Strukturansicht.

Strukturansicht

Aus den theoretischen Vorüberlegungen ging hervor, dass eine Darstellung, aus der die Struktur des virtuellen Instruments hervorgeht, notwendig ist. Diese Darstellung wurde mit der Strukturansicht realisiert. Damit die einzelnen *„Element“* Objekte ohne weitere Veränderungen in der Strukturansicht dargestellt werden können, implementiert die *„Element“* Klasse die *„TreeNode“* Schnittstelle (siehe Abb. 4.19). Klassen, die dieses Interface implementieren, werden über folgende Methoden, die zur automatisierten Darstellung benötigt werden, angesprochen: Die *„getParent“* Methode liefert den Vorfahren in der Struktur. In der *„isLeaf“* Methode wird überprüft, ob dieses Objekt Kinder hat. Mit *„getIndex“*, *„getChildCount“*, *„getChildAt“*, *„getAllowsChildren“* und *„children“* werden Methoden für den Zugriff auf die Kinder eines Objekts zur Verfügung gestellt.

Die Strukturansicht selbst ist in der *„ElementTree“* Klasse, die in Abb. 4.17 zu sehen ist, implementiert. Die bereits strukturiert vorliegenden Daten werden in dieser Klasse mit Hilfe der von Java in der *„Swing“* Bibliothek vorgegebenen *„JTree“* Klasse, die zur Darstellung baumartiger Strukturen gedacht ist, visualisiert. In dieser baumartigen Visualisierung werden die einzelnen Visualisierungselemente durch ihre Namen repräsentiert. Zusätzlich wird dem Namen des Visualisierungselementes, wenn es mit einem Element der Simulation verbunden ist, der Name des dazugehörigen Simulationselements vorangestellt.

Aus der *„ElementTree“* Klasse heraus können außerdem neue Kontextansichten für die in der Strukturansich selektierten Elemente erzeugt werden. Aus der Strukturansicht heraus wird die geometrische Ansicht gesteuert, wobei alle Strukturansichten hierbei auf eine einzelne geometrische Ansicht zugreifen. Über die Strukturansicht können die in der geometrischen Ansicht dargestellten Objekte selektiert werden. Nur die in der Strukturansicht gewählten *„Element“* Objekte werden in der geometrischen Ansicht mit allen ihren Unterobjekten dargestellt. Dieses Wechseln der darzustellenden Objekte erfolgt mit der in Abb. 4.17 gezeigten *„changeElement“* Methode der *„Instrument3d“* Klasse.

Kontextansicht

Die Kontextansicht wird in der bereits genannten *„ElementWindow“* Klasse erzeugt. Dieses Fenster setzt sich, wie aus Abb. 4.22 hervorgeht, aus drei einzelnen Informationsanzeigen zusammen. Diese Informationsanzeigen können, da sie mit der aus der *„Swing“* Bibliothek stammenden *„JTabbedPane“* Klasse angezeigt werden, von dem Benutzer wie bei einem Karteisystem über Reiter selektiert werden. Der Benutzer kann so zwischen drei Ansichten wählen:

1. Einer Ansicht, in der Informationen zu dem jeweiligen mechanischen Element und dessen Visualisierung angezeigt werden.

2. Einer Ansicht, in der die Daten des Motors, der das angezeigte Element antreibt, dargestellt sind.

3. Einer Ansicht, in der die Rasten des Elements aufgelistet werden.

Die Anzeigefläche, die für die Darstellung der elementbezogenen Daten genutzt wird, ist dementsprechend mit dem darzustellendem Element verbunden. Über diese Verbindung und die weiterleitende Verbindung aus Abb. 4.19 ist es möglich, auf die elementbezogenen Daten aus der Simulation zuzugreifen.

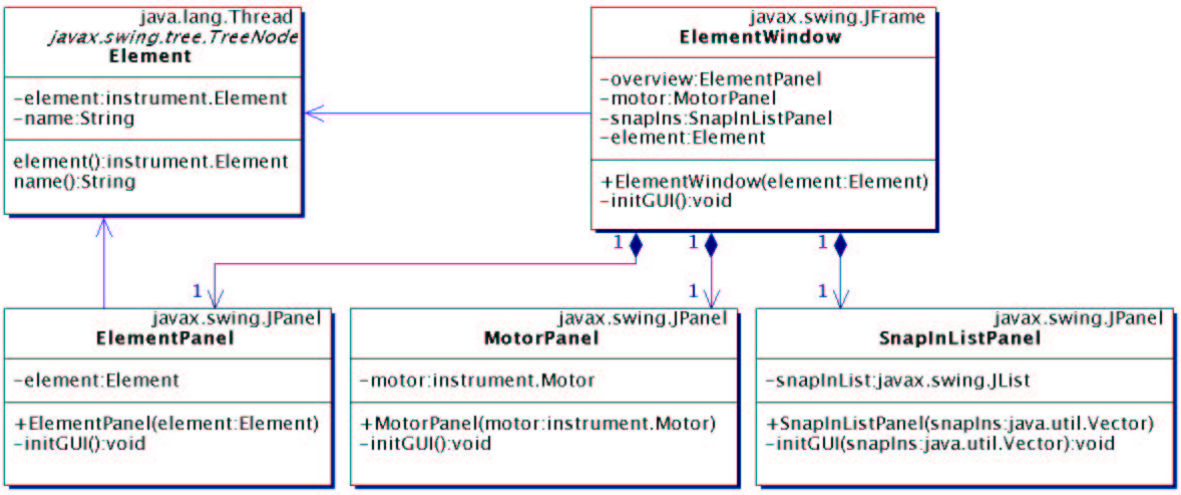


Abbildung 4.22: Übersicht der Klassen der Kontextansicht

In der Anzeigefläche, die den Motor und dessen Parameter visualisiert, besteht eine Verbindung zu dem entsprechenden *„Motor“* Objekt des *„instrument“* Paketes. Damit kann diese Anzeigefläche alle Objekte, die, wie in Abb. 4.5 gezeigt, mit dem *„Motor“* Objekt in Verbindung stehen, visualisieren.

Die letzte der drei genannten Anzeigeflächen dient der Visualisierung der Rasten, die auf dem entsprechenden Element angezeigt sind. Diese Rasten dienen, wie bereits beschrieben, dazu, Einrastvorgänge, wie sie bei dem späteren Element auftreten, zu simulieren. In dieser Anzeigefläche sollen sämtliche Rasten aufgelistet werden und es soll die Möglichkeit bestehen, Detailinformationen zu ausgewählten Rasten zu bekommen.

Die genauere Umsetzung der genannten drei Anzeigeflächen und die dazu nötige Objektinteraktion soll im folgenden beschrieben werden.

Elementdaten

Wird eine neue Kontextansicht zu einem Element erstellt, so zeigt das daraus resultierende Fenster zuerst die allgemeinen Elementdaten. Die für die Darstellung der Elementdaten zuständige *„ElementPanel“* Klasse setzt sich aus wiederum drei Anzeigeflächen zusammen (siehe Abb. 4.23).

Erster Bestandteil der *„ElementPanel“* Klasse ist die *„Element3dView“* Klasse, die dazu dient, das dreidimensionale Modell des selektierten Elements anzuzeigen. Dazu verfügt die *„Element3dView“* Klasse über einen eigenen dreidimensionalen Zeichenbereich, in dem ein eigenes virtuelles Universum erzeugt wird. Im Gegensatz zu der geometrischen Ansicht wird in diesem Visualisierungselement nur das Modell des ausgewählten Elements angezeigt. Eine nähere Beschreibung dieser Darstellung findet man bei Abb. 5.2 oder in der Dokumentation des virtuellen astronomischen Instruments.

Ein weiterer Bestandteil der *„ElementPanel“* Klasse ist die *„ElementDataPanel“* Klasse. Ihre Aufgabe besteht darin, Daten des *„Element“* Objekts aus dem *„instrument“* Paket darzustellen. Zu diesen Daten gehören etwaige positive oder negative Endschalter, die aktuelle Position, das Übertragungsverhältnis zwischen Motor und angetriebenem mechanischen Element sowie ein unter Umständen vorhandener Referrenzschalter und dessen Größe. Diese Daten werden mit Hilfe der *„LimitPanel“* Klasse dargestellt. Man sollte sich nicht von dem Namen dieser Klasse irritieren lassen. Die zu visualisierenden Daten werden vor ihrer Darstellung in *„Limit“* Objekt umgewandelt, damit bei einer Nichtexistenz das Datenfeld entsprechend graphisch

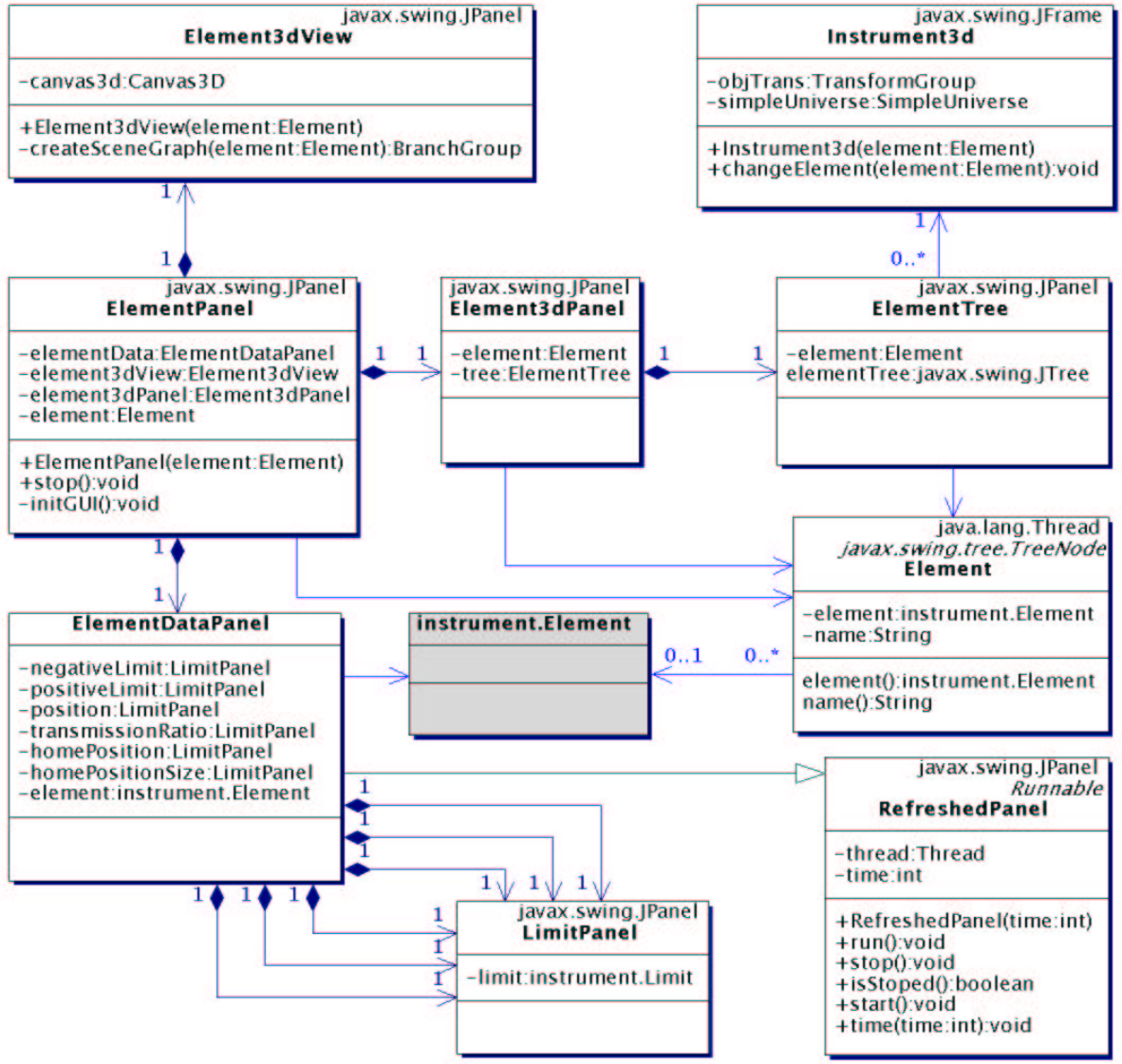


Abbildung 4.23: Visualisierung der Elementdaten

aufbereitet werden kann. Da „*Limit*“ Objekte des „*instrument*“ Pakets über die Eigenschaft „*exist*“ verfügen und sie somit neben ihrem Wert auch ihre Existenz widerspiegeln und außerdem ein Teil der darzustellenden Daten bereits in dieser Form vorliegt, empfiehlt sich diese Art der Darstellung.

Um den Aktualisierungsprozess der Daten der „*ElementDataPanel*“ Klasse zu verstehen, wird an dieser Stelle ein Konzept vorgestellt, das dazu verwendet wird, Daten nach gegebenen Zeitintervallen zu aktualisieren. Dieses Konzept wird noch an einigen anderen Stellen verwendet und ist für die Visualisierung von großer Bedeutung. Da für die Visualisierung in den theoretischen Vorüberlegungen gefordert wurde, dass die Daten der Simulation nicht beeinflusst werden dürfen, sind nur lesende Zugriffe auf die Daten der Simulation erlaubt. Außerdem findet der Datenaustausch zwischen Visualisierungsmodulen und der Simulation nur in eine Richtung statt. Aufgrund des Simulationsmodelles können Simulationsdaten immer nur auf Anfrage geliefert und nicht selbstständig generiert und an die Visualisierung weitergeleitet werden. Um eine Synchronität zwischen den Simulationsdaten und denen, die von der Visualisierung angezeigt werden, zu bieten, ist die „*RefreshedPanel*“ Klasse implementiert worden.

Jedes „*RefreshedPanel*“ Objekt besitzt einen eigenständigen Thread, der sich in einer Endlosschleife für eine in der Initialisierungsphase diese Objekts angegebene Zeit schlafen legt,

um danach die „*repaint*“ Methode dieses Objekts aufzurufen. Dieser Synchronisierungsprozess kann durch entsprechende Methoden angehalten, gestartet oder sein Zeitintervall verändert werden.

Damit eine Anzeigefläche wie die „*ElementDataPanel*“ Klasse diesen Synchronisierungsmechanismus benutzen kann, muss sie nur von der „*RefreshedPanel*“ Klasse erben und in einer eigenen „*paint*“ Methode festlegen, welche Daten wo angezeigt werden sollen. Durch den „*repaint*“ Aufruf des Threads der „*RefreshedPanel*“ Klasse wird dann diese „*paint*“ Methode aufgerufen und so die Daten aktualisiert. Die Aktualität der dargestellten Daten hängt somit nur von dem Zeitintervall ab. In Abb. 4.23 und Abb. 4.24 ist dieser Aktualisierungsprozess zu erkennen.

Als letzter der drei Anzeigebereiche, aus denen sich die „*ElementPanel*“ Klasse zusammensetzt, bleibt die „*Element3dPanel*“ Klasse zu nennen. Diese Klasse dient nicht, wie ihr Name vermuten lässt, der Anzeige der dreidimensionalen Ansicht, sondern wird zur Anzeige der Struktur der dreidimensionalen Visualisierungselemente benutzt. In diesem Anzeigebereich wird der Name des Visualisierungselements sowie dessen Vorfahre und seine Kinder angezeigt. Diese Anzeige funktioniert durch die bereits beschriebene „*ElementTree*“ Klasse der Strukturansicht. Somit besitzt eine Kontextansicht die Möglichkeit, die geometrische Ansicht zu steuern oder eine neue Kontextansicht zu öffnen.

Neben den allgemeinen Elementdaten müssen auch die Motordaten visualisiert werden. Dies geschieht in Form der „*MotorPanel*“ Klasse.

Motordaten

Für die Visualisierung der Motordaten wird die „*MotorPanel*“ Klasse benutzt, die sich ebenfalls, wie die „*ElementPanel*“ Klasse, aus anderen Klassen zusammensetzt (siehe Abb. 4.24). Die Aufgabe der „*MotorPanel*“ Klasse besteht darin, diese sechs Anzeigen entsprechend anzuordnen und bei Beendigung der Darstellung ihre Synchronisierungsthreads anzuhalten. Die sechs Unteransichten werden durch die „*MotorAnglePanel*“ Klasse, die „*MotorParameterPanel*“ Klasse, die „*MotorSimulationPanel*“ Klasse, die „*MotorSpeedPanel*“ Klasse sowie die „*MotorRealValuePanel*“ Klasse und die „*MotorExpectedValuePanel*“ Klasse gebildet. Aus Abb. 4.24 ist zu erkennen, welche der Klassen die „*RefreshedPanel*“ Klasse benutzen, um ihr Daten entsprechend zu synchronisieren. Im folgenden sollen die genannten Unteransichten genauer erläutert werden.

Als erstes werden die „*MotorRealValuePanel*“ Klasse und die „*MotorExpectedValuePanel*“ Klasse genauer beschrieben. Beide Klassen dienen dazu, den aktuellen Stellungswinkel des Motors, seine Geschwindigkeit, sowie seine Beschleunigung anzuzeigen. Diese beiden Klassen unterscheiden sich nur darin, dass die eine Klasse die von der Simulation berechneten tatsächlichen Werte anzeigt, während die andere Klasse die ebenfalls von der Simulation berechneten, jedoch von der Elektronik erwarteten Werte anzeigt. Dieses Anzeigen geschieht mit Hilfe der „*JTextField*“ Klasse der „*Swing*“ Bibliothek, die ein einfaches Textfeld zur Verfügung stellt. Zur besseren optischen Hervorhebung wird außerdem die Hintergrundfarbe der Textfelder verändert, während sich der Motor bewegt. Der genaue Unterschied zwischen tatsächlichen und erwarteten Werten, sowie der Grund für deren Asynchronität ist in dem Kapitel 3 bereits beschrieben worden.

Ausschlaggebend für diese Wertunterschiede sind die bei der Simulation vorgestellten Simulationsparameter. Diese werden mit Hilfe der „*MotorSimulationPanel*“ Klasse dargestellt. Die maximal zulässige Beschleunigung, die maximale Geschwindigkeit sowie die Zufallsfunktion, die bei dem Überschreiten der maximalen Geschwindigkeit diese moduliert, werden in einfachen Textfeldern ausgegeben.

Die „*MotorParameterPanel*“ Klasse dient der Visualisierung der wichtigsten Motorparame-

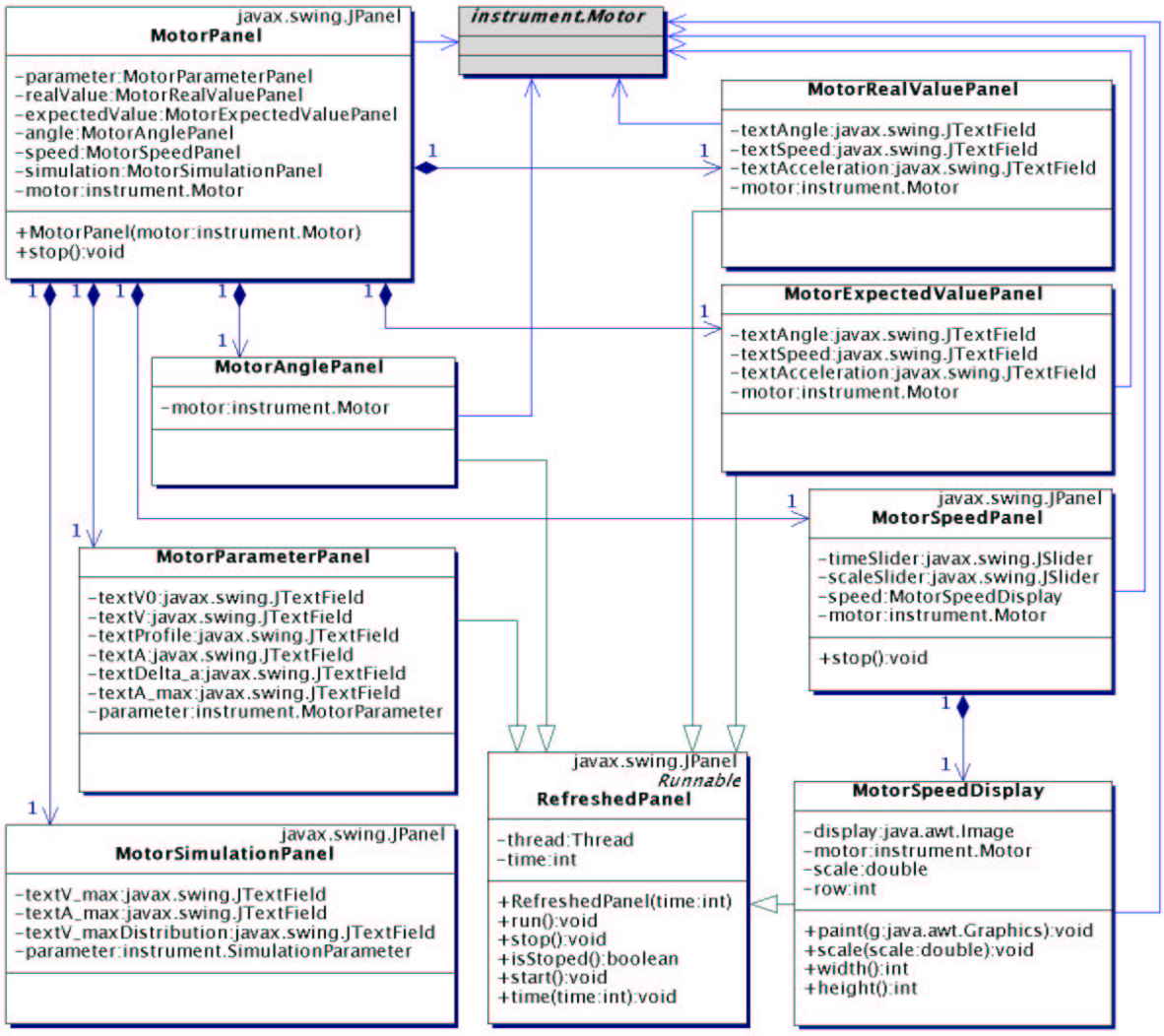


Abbildung 4.24: Klassen zur Visualisierung der Motordaten

ter. Obwohl diese Informationen auch über die Firmware abzufragen sind, werden die wichtigsten Parameter in dieser Anzeigefläche ausgegeben. Als wichtigste Parameter werden die Startgeschwindigkeit, die gewünschte Geschwindigkeit, die Startbeschleunigung, die Beschleunigungsänderung, die maximale Beschleunigung und das Profil einer Fahrbewegung ausgewählt. Dieses Anzeigen geschieht ebenfalls in Textfeldern, wobei deren Hintergrundfarbe die von dem selektierten Fahrprofil benötigten Parameter hervorhebt.

Die letzten beiden noch nicht besprochenen Anzeigeklassen stellen ihren Inhalt, im Gegensatz zu den bereits genannten, auf graphische Art dar. Die *„MotorAnglePanel“* Klasse dient der Darstellung des tatsächlichen und des erwarteten Stellungswinkels. Dies geschieht durch einfaches Nutzen der graphischen Fähigkeiten von Java (siehe [Knu99]). Der Stellungswinkel des Motors wird symbolisch durch einen Kreis mit einer winkelweisenden Linie repräsentiert (siehe die Beschreibung in Kapitel 5). Damit der tatsächliche und der erwartete Winkel des Motors aktualisiert werden, erbt die *„MotorAnglePanel“* Klasse alle Aktualisierungsmöglichkeiten von der *„RefreshedPanel“* Klasse.

Die *„MotorSpeedPanel“* Klasse ist die umfangreichste Klasse, die zur Visualisierung der Motordaten verwendet wird. Ihre Aufgabe besteht darin, die Motorgeschwindigkeit in einer Art Fahrtenschreiber darzustellen. Dabei soll es dem Benutzer ermöglicht werden, die Geschwindigkeit, mit der der Fahrtenschreiber aufzeichnet, sowie seine Auflösung einzustellen. Das Aktualisierungsintervall sowie die Auflösung werden in der *„MotorSpeedPanel“* Klasse durch die

aus der „*Swing*“ Bibliothek stammenden „*JSlider*“ Objekte eingestellt. Die Anzeige der Geschwindigkeit selbst findet in der externen „*MotorSpeedDisplay*“ Klasse statt. Dazu benutzt die „*MotorSpeedDisplay*“ Klasse ein „*Image*“ Objekt aus der zu Java gehörenden „*AWT*“ Bibliothek. Damit wird es möglich, die Geschwindigkeit in den Fahrtenschreiber einzuzeichnen, bevor dieser dargestellt wird und erst danach die aktuelle Geschwindigkeitskurve mit der angezeigten Kurve zu tauschen. Dieses Verfahren wird „*doublebuffering*“ genannt und verhindert ein Flackern, bei der Aktualisierung der Motorgeschwindigkeit (siehe [Knu99]). Der Darstellungsprozess erfolgt fortlaufend, so dass sich eine Geschwindigkeitskurve ergibt. Das Sequenzdiagramm aus Abbildung 4.25 stellt diesen Prozess dar.

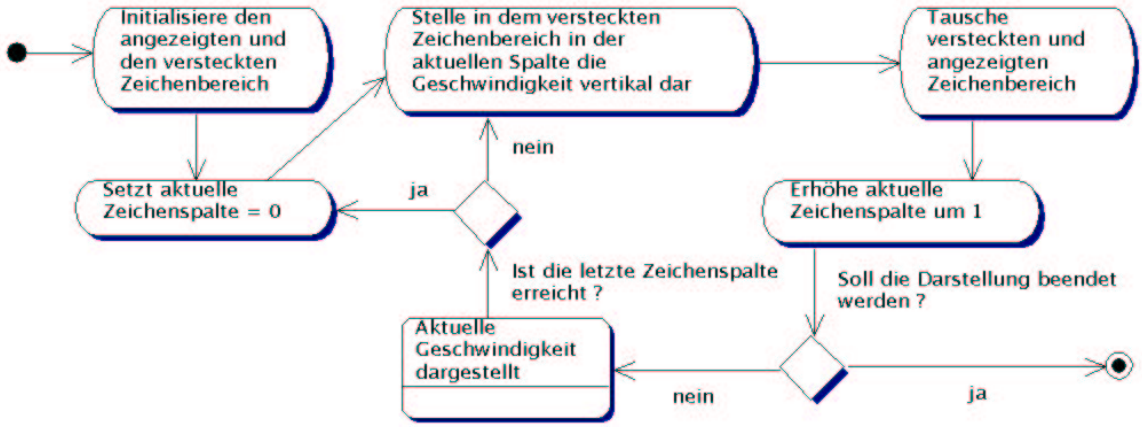


Abbildung 4.25: Funktionsweise des Fahrtenschreibers

In Kapitel 5 werden Darstellungsart und Interpretation der Graphik näher beschrieben. Als letzter der drei Bereiche der „*ElementWindow*“ Klasse aus Abb. 4.22 wird die „*SnapInListPanel*“ Klasse, die zur Darstellung der Rasten benötigt wird, beschrieben.

Rasten

Die aus der Simulation bekannten Rasten werden durch die „*SnapInListPanel*“ Klasse dargestellt. Ein Objekt dieser Klasse verfügt zum einen über eine Liste der an dem dargestellten Element vorhandenen Rasten, zum anderen über eine Detailansicht. Die Listendarstellung benutzt die aus der „*Swing*“ Bibliothek stammende „*JList*“ Klasse, um die bei der Erzeugung der Anzeigefläche in Form eines Vektors übergebenen Rasten anzuzeigen. Die Detaildarstellung einer in der Liste selektierten Raste geschieht durch die „*SnapInPanel*“ Klasse (siehe Abb. 4.26). Dazu beinhaltet diese Klasse zwei weitere Klassen. Die „*ModificationParameterPanel*“ Klasse wird benutzt, um Bewegungsparameter anzuzeigen. Diese sind ausschlaggebend für das Verhalten des Elementes, wenn es sich innerhalb der vorgegebenen Positionen befindet. Neben den Bewegungsparametern sind einige bereits genannte Positionen für eine Raste wichtig. Diese sind Abb 4.5 zu entnehmen. Die Darstellung der Positionen erfolgt auf die gleiche Art und Weise wie die Darstellung der Elementdaten in der „*ElementDataPanel*“ Klasse (vergleiche Abb. 4.23). Durch die „*snapIn*“ Methode der „*SnapInPanel*“ Klasse ist es möglich, die dargestellte Raste zu wechseln. Diese Methode benutzt die „*SnapInListPanel*“ Klasse, um die Ansicht mit den selektierten Rasten zu synchronisieren.

Nachdem die Umsetzung der Kontextansicht beschrieben wurde, bleibt die Erläuterung der Umsetzung der geometrischen Ansicht.

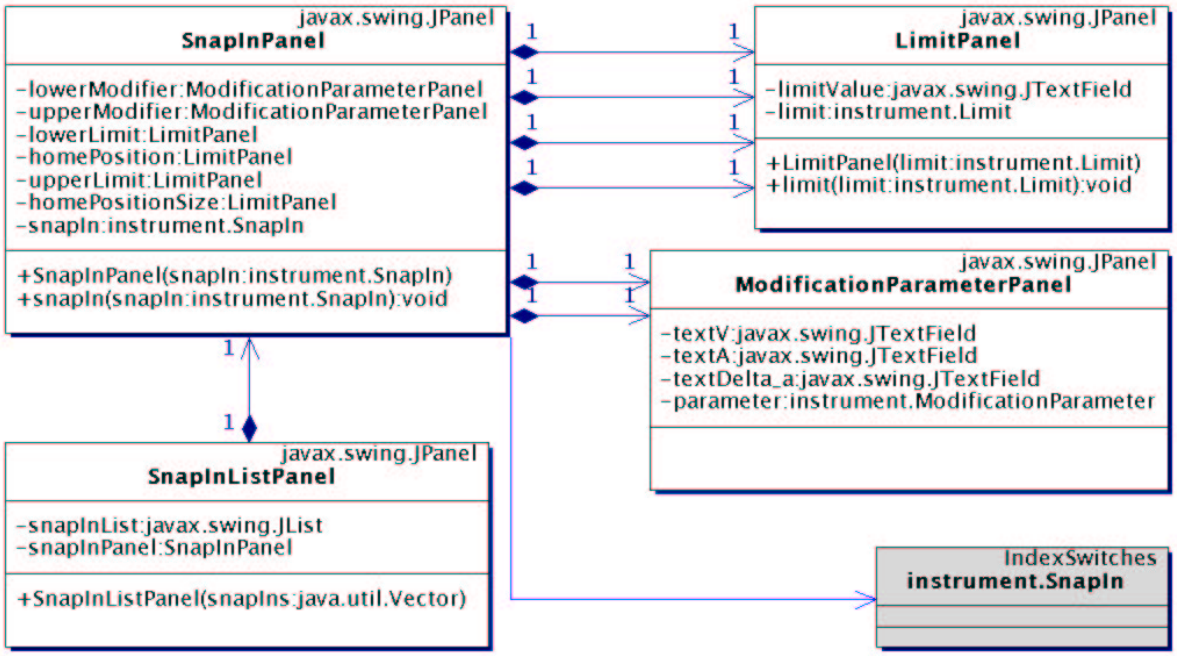


Abbildung 4.26: Visualisierung der Rasten

Geometrische Ansicht

Aufgabe der geometrischen Ansicht ist es, das virtuelle Instrument dreidimensional darzustellen. Dies geschieht durch die in Abb. 4.17 gezeigte *„Instrument3d“* Klasse. In einem Objekt dieser Klasse wird ein virtuelles Universum erzeugt, welches in einem speziellen Zeichenbereich des Objekts angezeigt wird. In dieses Universum wird der Szenenstruktur entsprechend ein Ortsobjekt eingefügt (siehe Abb. 4.18). Unter diesem Ortsobjekt befindet sich ein Transformationsknoten, der die Mausbewegungen entsprechend umsetzt. Dadurch wird es möglich, die dargestellten Objekte mit Hilfe der Maus zu drehen, zu verschieben oder zu vergrößern. Damit dieser Transformationsknoten sich auf die Objekte auswirken kann, ist die Wurzel der darzustellenden Szene direkt mit ihm verbunden. Dies geschieht mit der in Abb. 4.21 gezeigten Verknüpfungsmethode.

Dadurch, dass die in der Datenstruktur aus Abb. 4.19 gezeigte *„Element“* Klasse die Möglichkeit bietet, direkt das zugehörige Modell mit allen Unterelementen zu erhalten, muss sich um die Szenenkonstruktion nicht weiter gekümmert werden. Aus dem leicht gekürzten Quellcode von Algorithmus 1 geht die Einfachheit der Verwendung von Java3d hervor.

Zuerst wird ein einfacher dreidimensionaler Zeichenbereich erstellt. Nachdem die Wurzel der Szene erzeugt und das Entnehmen von Elementen aus ihr erlaubt wurde, wird ein Transformationsknoten eingefügt. Die näheren Angaben zur Umsetzung der Mausbewegung sowie die Spezifikation der Lichtquellen sind herausgekürzt. Danach wird die entsprechende Methode des darzustellenden Objekts aufgerufen und der zurückgegebene Szenengraph an den Transformationsknoten angefügt. Zum Schluss wird die Wurzel der gesamten Szene in das neu erzeugte virtuelle Universum eingefügt.

Nachdem die Umsetzung der Strukturansicht als auch der Kontext- und der geometrischen Ansicht beschrieben worden sind, wird als nächstes die Umsetzung der theoretischen Vorüberlegungen zu der Benutzungsoberfläche behandelt.

Algorithm 1 Programmzeilen der *"Instrument3d"* Klasse

```
canvas3d = new Canvas3D(SimpleUniverse.getPreferredConfiguration());
objRoot = new BranchGroup(); //erzeuge Wurzel der Szene
objRoot.setCapability(BranchGroup.ALLOW_DETACH); //Elementwechsel erlauben
objTrans = new TransformGroup();
.....
..... //Mausbewegungen mit Transformationsknoten verbinden
.....
objRoot.addChild(objTrans);
.....
..... //Hintergrund und Lichtquellen erzeugen
.....
if (element!=null) objTrans.addChild(element.modelWithChilds());
simpleUniverse = new SimpleUniverse(canvas3d); //erzeuge virtuelles Universum
simpleUniverse.addBranchGraph(objRoot); //füge Szene ein
```

4.5.3 Benutzungsoberfläche

Die Benutzungsoberfläche selbst ist auf die einzelnen Fenster der verschiedenen Visualisierungsmodule verteilt worden. Zentrales Element der Benutzungsoberfläche stellt die *„Main-Window"* Klasse dar (siehe Abb. 4.17). Aus diesem Fenster heraus können, wie in der theoretischen Vorüberlegung bereits genannt, die unterschiedlichen Ansichten sowie das Visualisierungsmodul für die Kommunikationsdaten gestartet werden. Damit verteilt sich die Komplexität der Bedienung auf die unterschiedlichen Ansichten, da Programmfunktionen nur in den Ansichten zur Verfügung stehen, in denen sie Verwendung finden.

Dadurch, dass dem Benutzer die Möglichkeit gegeben wird, die von ihm benötigten Daten durch Auswahl der Ansichten selbst zu selektieren, ist eine weitere Forderung der Vorüberlegungen erfüllt.

Die von [Str98] für eine gute Visualisierung strukturierter geometrischer Daten geforderte Dreiteilung ist der Kern der Benutzungsoberfläche. Durch die Umsetzung der Dreiteilung der Visualisierung der Mechanik in Strukturansicht, Kontextansicht und geometrische Ansicht wird nun ein sehr intuitiver Umgang mit den Modelldaten des virtuellen Instruments ermöglicht.

Ein weiterer Gesichtspunkt, der bei den theoretischen Vorüberlegungen zu einer Benutzungsoberfläche genannt wurde, war ein in das Programm integriertes Hilfesystem. Hierbei wurde sich auf eine einfache Hilfe in Form von *„tool tips"* beschränkt. Jedem Funktionselement der Benutzungsoberfläche wurde durch einen *„tool tip"* ein kleiner Informationstext hinzugefügt, der mit wenigen Worten die Funktion eines Bedienelements beschreibt. In Java ist das Einfügen solcher Informationstexte bereits vorgesehen und erfordert somit keinen weiteren Programmieraufwand.

Nachdem die Umsetzung der Vorüberlegungen und somit die Implementierung des virtuellen astronomischen Instruments mit Hilfe der Programmiersprache Java beschrieben wurde, soll in dem nächsten Kapitel der so entstandene Softwareprototyp vorgestellt werden.

Ein Problem wird nicht im Computer gelöst, sondern in irgendeinem Kopf. Die ganze Apparatur dient nur dazu, diesen Kopf so weit zu drehen, dass er die Dinge richtig und vollständig sieht.

Charles Kettering

Kapitel 5

Lucifer VR

In diesem Kapitel wird das virtuelle astronomische Instrument vorgestellt; besser gesagt: Der Softwareprototyp, mit dessen Hilfe ein virtuelles Instrument erstellt werden kann. Die entwickelte Software trägt den Namen *„LuciferVR“*, wobei das *„VR“* für virtuelle Realität steht.

Da zu dem jetzigen Zeitpunkt noch nicht alle erforderlichen Modelldaten zu dem Lucifer Instrument vorhanden sind, wird in diesem Kapitel anstelle eines astronomischen Instruments eine bereits existierende Testeinheit simuliert und visualisiert. Diese Testeinheit besteht aus einer Steuereinheit, die die entsprechende Elektronik enthält, sowie einem Testaufbau, der aus einem Motor, zwei Endschaltern und einer Kalibrierungssonde besteht. Diese Testeinheit wurde vom Max Planck Institut für Astronomie in Heidelberg gefertigt. Dabei entspricht die verwendete Elektronik der Steuerelektronik, die für das Lucifer Instrument benutzt wird. Dadurch ist, bis auf die Anzahl von Motoren, die Testeinheit mit dem Lucifer Instrument vergleichbar. Sobald sämtliche Modelldaten des Lucifer Instruments vorhanden sind, kann das virtuelle astronomische Instrument von der Software nachgebildet werden und so die Entwicklung der Steuersoftware unterstützen.

Bevor die *„LuciferVR“* Software beschrieben wird, werden zunächst die Hardwareanforderungen sowie die Installation der Software behandelt. Darauf folgt eine Beschreibung der Benutzungsoberfläche des Programms. Zuletzt werden die Ergebnisse aus dem Vergleich zwischen reeller und virtueller Testeinheit betrachtet.

5.1 Hardwareanforderung und Installation

Wie in Kapitel 4 bereits beschrieben wurde, ist der Softwareprototyp des virtuellen astronomischen Instruments in Java programmiert worden. Dadurch ergibt sich, dass keine spezielle Rechnerarchitektur für die Ausführung der Software nötig ist. Vielmehr spielt die Rechenleistung der verwendeten Hardware eine Rolle. Der Rechner, der für die Installation des virtuellen Instruments vorgesehen ist, sollte im mittleren Leistungsbereich heutiger PC’s liegen und über eine Graphikkarte mit 3D-Unterstützung verfügen. Geringere Leistung führt zu einer langsameren Visualisierung, ohne jedoch merklich die Simulation zu beeinträchtigen.

Um das virtuelle Instrument auf einem zuvor gewählten Rechner laufen zu lassen, ist es notwendig, die entsprechende virtuelle Maschine von Java zu installieren. Diese ist unter `java.sun.com` zu erhalten, wobei eine Version ab JRE 1.3.1 (*java runtime environment*) empfohlen wird. Außerdem ist es nötig die Java3d Bibliothek zu installieren, da diese nicht zu dem Lieferumfang von Java gehört. Die Java3d Bibliothek ist ebenfalls unter `java.sun.com` zu erhalten. Für nicht auf den Internetseiten von Sun unterstützte Rechnerplattformen findet man die entsprechenden Downloadmöglichkeiten leicht bei einer Suche unter `www.google.com`. Die entsprechenden Installationshinweise liegen der virtuellen Maschine von Java sowie der Java3d Softwarebibliothek bei und sind entsprechend zu beachten.

Die Installation des virtuellen Instruments geschieht auf recht einfache Weise. Nachdem ein

entsprechendes Verzeichnis für die Software neu angelegt wurde, müssen nur die Dateien des virtuellen Instruments in dieses kopiert werden.

Danach kann mit dem Aufruf, der in dem entsprechenden Verzeichnis geschehen sollte, von „`java -Xms256M -Xmx512M luciferVR`“ das virtuelle Instrument gestartet werden. Dabei wird mit dem „`-Xms`“ Parameter der für das Programm zu Beginn zur Verfügung gestellte Arbeitsspeicher auf 256 Megabyte voreingestellt. Mit dem „`-Xmx`“ Parameter wird der maximal nutzbare Arbeitsspeicher auf 512 Megabyte beschränkt. Diese Werte können dem Rechner entsprechend angepasst werden, wobei zu bedenken ist, dass Java für die dreidimensionale Darstellung relativ viel Arbeitspeicher benötigt.

Nach erfolgreichem Programmstart erscheint die Benutzungsoberfläche in Form des sehr übersichtlich gehaltenen Hauptfensters.

5.2 Benutzungsoberfläche

Die Benutzungsoberfläche erscheint, im Gegensatz zu vielen anderen Programmen, sehr übersichtlich. Dies kommt daher, dass der Benutzer erst durch Selektion einzelner zu visualisierender Elemente mit den entsprechenden Programmfunktionen konfrontiert wird (vergleiche Kapitel 3 und Kapitel 4). Anfangs zeigt sich dem Benutzer nur das in Abb. 5.1 gezeigte Hauptfenster.

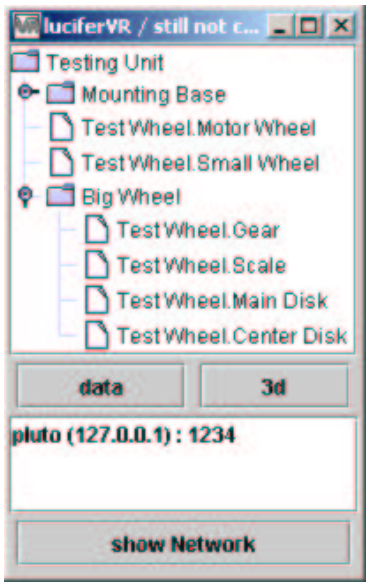


Abbildung 5.1: Hauptfenster mit Struktur- und Kommunikationsansicht

In Abb. 5.1 ist zum einen die Strukturansicht dargestellt, aus der sich die Kontext- und geometrische Ansicht aufrufen lassen, zum anderen ist eine Liste der verwendeten Netzwerkadressen abgebildet. Aus dieser Liste der Netzwerkadressen kann der Benutzer selbst auswählen, zu welcher Netzwerkadresse er die Kommunikationsdaten visualisiert haben möchte. Nach Selektion des entsprechenden Listeneintrags und Betätigung des *„show network“* Knopfes erscheint ein entsprechendes Fenster, das die Daten der Netzwerkverbindung anzeigt.

Zunächst wird die in Abb. 5.1 dargestellte Strukturansicht näher beschrieben.

5.2.1 Strukturansicht

Die Strukturansicht, die sowohl in dem Hauptfenster als auch in der in Abb. 5.2 gezeigten Elementenansicht enthalten ist, dient der Visualisierung der Komponentenverknüpfungen. Die in

Abb. 5.1 gezeigte Struktur gehört zu der bereits erwähnten Testeinheit. Wie leicht zu erkennen ist, setzt sich diese Testeinheit aus einer Montageeinheit, zwei einzelnen Rädern und einer Radeinheit zusammen. Aus den Piktogrammen, die den Einträgen der Strukturansicht vorangestellt sind, ist zu erkennen, dass sich die Montage- sowie die Radeinheit aus einzelnen Elementen zusammensetzen. Dabei ist die Unterstruktur der Radeinheit angezeigt, während die Unterelemente der Montageeinheit verborgen bleiben. Wie detailliert die Struktur dargestellt wird, bleibt der Auswahl des Benutzers vorbehalten. Aus der Strukturansicht geht außerdem hervor, welche Visualisierungselemente mit welchen Simulationselementen verknüpft sind. Dazu wird den Visualisierungselementen der Name der visualisierten Simulationskomponente vorrangestellt. So zeigt z.B. der Struktureintrag *„TestWheel.MotorWheel“* an, dass das angezeigte Motorenrad zur Visualisierung des simulierten Testrades genutzt wird. Da die verwendete Testeinheit nur über einen Motor verfügt, der jedoch einige Elemente antreibt, sind in der Strukturansicht mehrere Visualisierungskomponenten mit diesem Motor verbunden.

Nach Selektion eines Eintrags in der Strukturansicht kann zu diesem die geometrische Ansicht gewählt werden. Dies geschieht durch drücken des *„3d“* Knopfes. Um die Kontextansicht zu erhalten, muss ein Element, das auch über eine entsprechende Simulationskomponente verfügt, angewählt werden. Durch Anwählen des *„data“* Knopfes erscheint die Kontextansicht. Dieses Visualisieren von Simulationsdaten funktioniert nur bei Elementen, die über entsprechende Verknüpfungen verfügen.

Als nächstes wird die soeben genannte Kontextansicht näher beschrieben.

5.2.2 Kontextansicht

In der Titelleiste des Fensters, das die Kontextansicht beinhaltet, wird der Name des dargestellten Simulationselements angezeigt. Die Kontextansicht selbst setzt sich, wie bereits in Kapitel 4 beschrieben, aus drei Unteransichten zusammen. Diese sind über die in den Abb. 5.2, 5.3 und 5.4 zu erkennenden Reiter am oberen Rande des Fensters auszuwählen.

Elementdaten

Wenn man die Elementdaten zu sehen wünscht und durch den entsprechenden Reiter selektiert, erhält man die in Abb. 5.2 gezeigte Ansicht. Der darunterliegende Anzeigebereich des Fensters unterteilt sich in drei Regionen.

Die erste und oben liegende Region beinhaltet allgemeine Elementdaten. Dazu gehören eventuell vorhandene Positionen des positiven und negativen Endschalters sowie die aktuelle Elementposition zwischen diesen Endschaltern. Die Positionen werden in Grad, bezogen auf die Winkelausrichtung des antreibenden Motors, angegeben. Sie entsprechen der Motorstellung, modifiziert mit dem ebenfalls dargestellten Übertragungsverhältnis zwischen Motor und Element. So sagt das angezeigte Übertragungsverhältnis von *„8.00“* aus, dass acht Motorumdrehungen einer Elementumdrehung entsprechen. Außerdem werden in dieser Anzeigeregion noch die Position eines evtl. vorhandenen Referenzschalters sowie seine Breite angegeben.

Die zweite Region, die in der linken unteren Ecke des Fensters liegt, wird zum Anzeigen des dreidimensionalen Modells des Visualisierungselements benutzt. Wie bereits in Kapitel 4 beschrieben, handelt es sich hierbei um ein eigenständiges geometrisches Visualisierungselement. Dieses Visualisierungselement ist durch Benutzung der Maus zu beeinflussen. Bei gedrückter linker Maustaste werden Rotationsbewegungen ausgeführt. In Kombination mit der rechten Maustaste führen deren Bewegungen zu einer Translation des dargestellten Modells. Wird neben der linken Maustaste auch die *„alt“*- Taste der Tastatur benutzt, kann die Szene vergrößert bzw. verkleinert werden. In Abb. 5.2 ist das Zahnrad, welches in der Strukturansicht aus Abb. 5.1 als *„TestWheel.Gear“* bezeichnet ist, dargestellt.

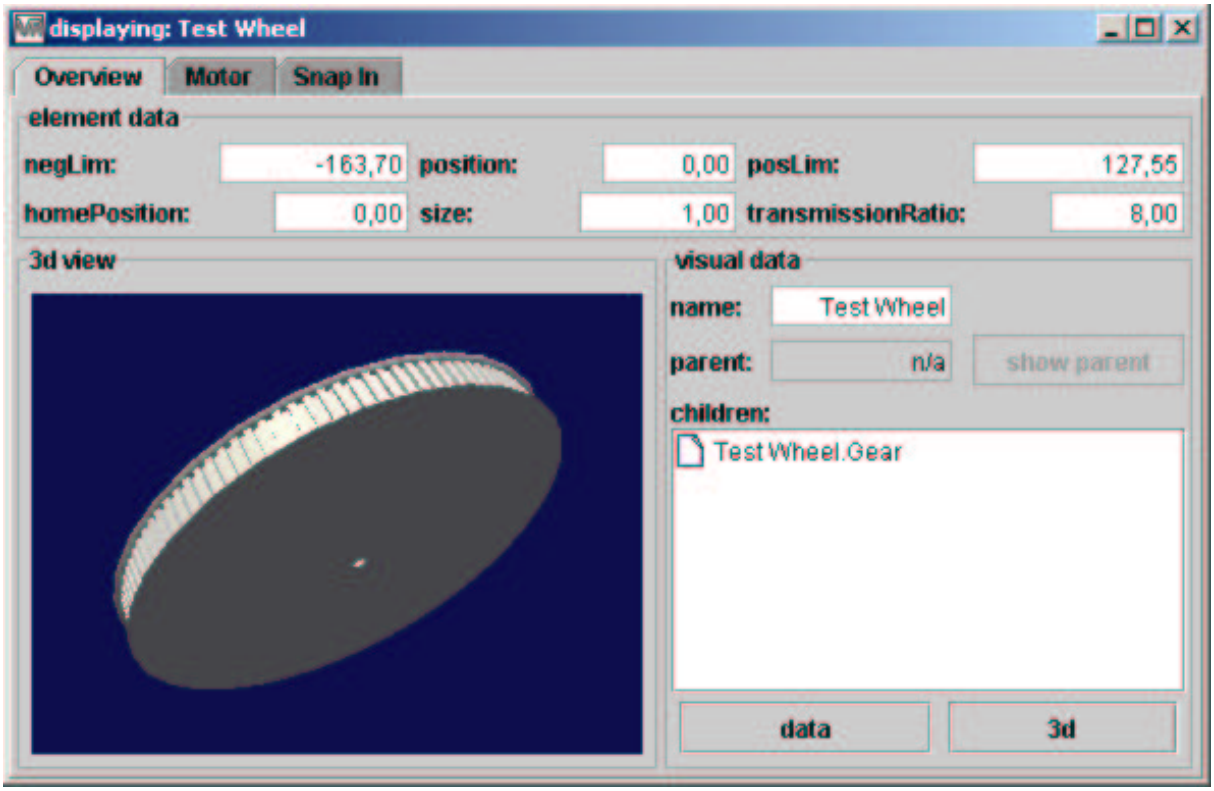


Abbildung 5.2: Elementdaten

In der letzten Anzeigeregion, am rechten unteren Rand, werden die Verknüpfungen des Visualisierungselements dargestellt. So besteht zum einen die Möglichkeit, eine Kontextansicht zu evtl. vorhandenen Vorgängern zu öffnen, zum anderen werden alle mit diesem Element verknüpften Unterelemente in einer eigenen Strukturansicht, die bereits beschrieben wurde, dargestellt.

Motordaten

Wird über den entsprechenden Reiter die Ansicht zu den Motordaten selektiert, erscheinen diese wie in Abb. 5.3 dargestellt. Damit die Daten übersichtlich angezeigt werden, ist die Visualisierung der Motordaten in sechs Unterbereiche aufgeteilt.

Der mitunter größte Bereich wird dabei von dem in Kapitel 4 vorgestellten Fahrtenschreiber eingenommen. Dieser dient der Darstellung der Motorgeschwindigkeit. Hierbei können sowohl die Zeit - mit der die Geschwindigkeitskurve dargestellt wird - als auch die maximal angezeigte Geschwindigkeit verändert werden. Dies geschieht durch parallel zu den zu verändernden Achsen verlaufende Schieber. Die aktuelle Position des Schreibkopfes der Geschwindigkeitsanzeige wird durch eine graue Linie angezeigt. Da die Darstellung der Motorgeschwindigkeit fortlaufend geschieht und somit alte Daten überschrieben werden, kann die Anzeige durch einen Mausklick angehalten bzw. wieder gestartet werden. Die Geschwindigkeit des Motors, die in Schritte pro Sekunde ausgegeben wird, wird vertikal aufgetragen. Dabei bedeuten rote Farben ein den Winkel des Motors verringernde Bewegungsrichtung und grüne eine entsprechend den Winkel Erhöhende. Durch den Fahrtenschreiber werden sowohl die reellen Geschwindigkeiten als auch die von der Elektronik erwartete dargestellt [1]. Hierbei repräsentiert der hellere Farbton die reelle Geschwindigkeit und der dunklere die Erwartete.

In Abb. 5.3 ist das Fehlverhalten, das durch Überschreiten der maximalen Motorgeschwindigkeit entsteht, zu erkennen. Im Plateaubereich der dargestellten Geschwindigkeitskurve stellt

[1] für Unterschied zwischen reeller und erwarteter Geschwindigkeit siehe Kapitel 3

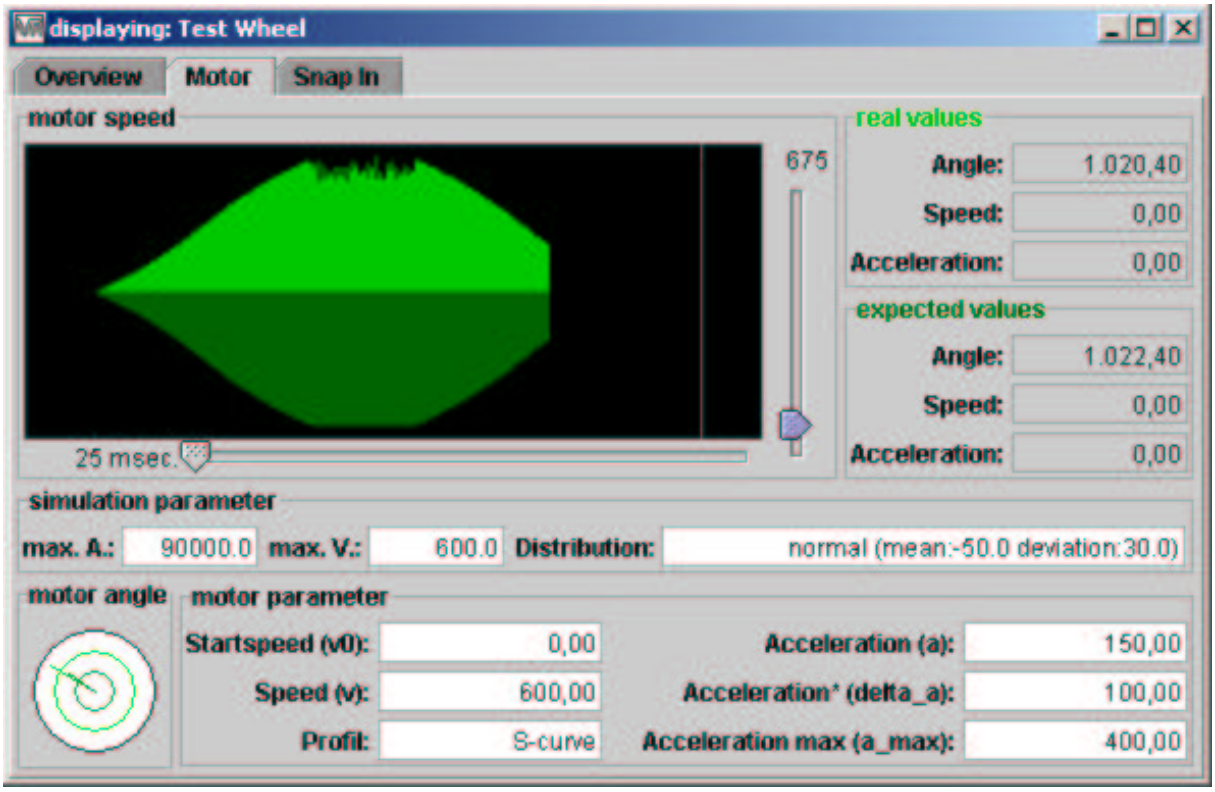


Abbildung 5.3: Motordaten

man bei der reellen Geschwindigkeit ein gewisses Rauschen[2] fest, was jedoch nicht bei der erwarteten Geschwindigkeit zu sehen ist. Am Ende der Kurve ist das Erreichen des positiven Endschalters zu erkennen, welches durch den abrupten Geschwindigkeitsabfall erkennbar wird.

Rechts neben dem Fahrtenschreiber werden der Winkel, die Geschwindigkeit sowie die Beschleunigung des Motors textuell angezeigt. Dabei wird auch wieder zwischen reellen und erwarteten Werten unterschieden. Führt der Motor eine Bewegung aus, werden diese Textfelder farblich hervorgehoben, indem der Hintergrund weiß gefärbt wird. Für z.B. den Fall, dass sich der Motor nur aus Sicht der Elektronik bewegt und somit reell keine Bewegung ausführt, werden auch nur die Textfelder der erwarteten Werte farblich hervorgehoben.

Unter dem Fahrtenschreiber und den textuellen Wertanzeigen befindet sich eine Zone, die zur Darstellung von drei Parametern, die die Motorsimulation betreffen, genutzt wird. Diese Parameter sind ausschlaggebend für das Geschwindigkeitsverhalten. Dabei gibt der mit *„max. A.“* gekennzeichnete Parameter die Grenze a_{sim} an, ab der die Beschleunigung ein zu hohes Drehmoment vom Motor erfordert und somit zu einem Ausbleiben der reellen Bewegung führt. Der mit *„max. V.“* bezeichnete Parameter bestimmt, ab welcher Geschwindigkeit v_{sim} das in Abb. 5.3 gezeigte Rauschen auftritt. Ausschlaggebend für dieses Rauschen ist die Zufallsverteilung[3], die dem letzten Textfeld entnommen werden kann.

In der unteren linken Ecke der Motordatendarstellung werden die beiden Winkel mittels einer einfachen Graphik dargestellt. Hier repräsentiert die hellere Farbe wiederum den reellen und die dunklere den erwarteten Winkel. Die Winkel werden mittels der Modulo Operation auf einen Kreis zwischen 0 und 360 Grad projiziert, wobei 0 Grad einer Zeigerstellung von 0 Uhr entspricht. Die Winkel werden dem Uhrzeigersinn folgend in diesem Kreis dargestellt.

Schließlich befindet sich in der unteren rechten Ecke des Fensters ein Anzeigebereich, in dem die wichtigsten Motorparameter dargestellt werden. Hierbei werden die Parameter die für

[2] vergleiche Kapitel 3 und Kapitel 4

[3] Die verwendbaren Zufallsfunktionen werden in Kapitel 4 vorgestellt.

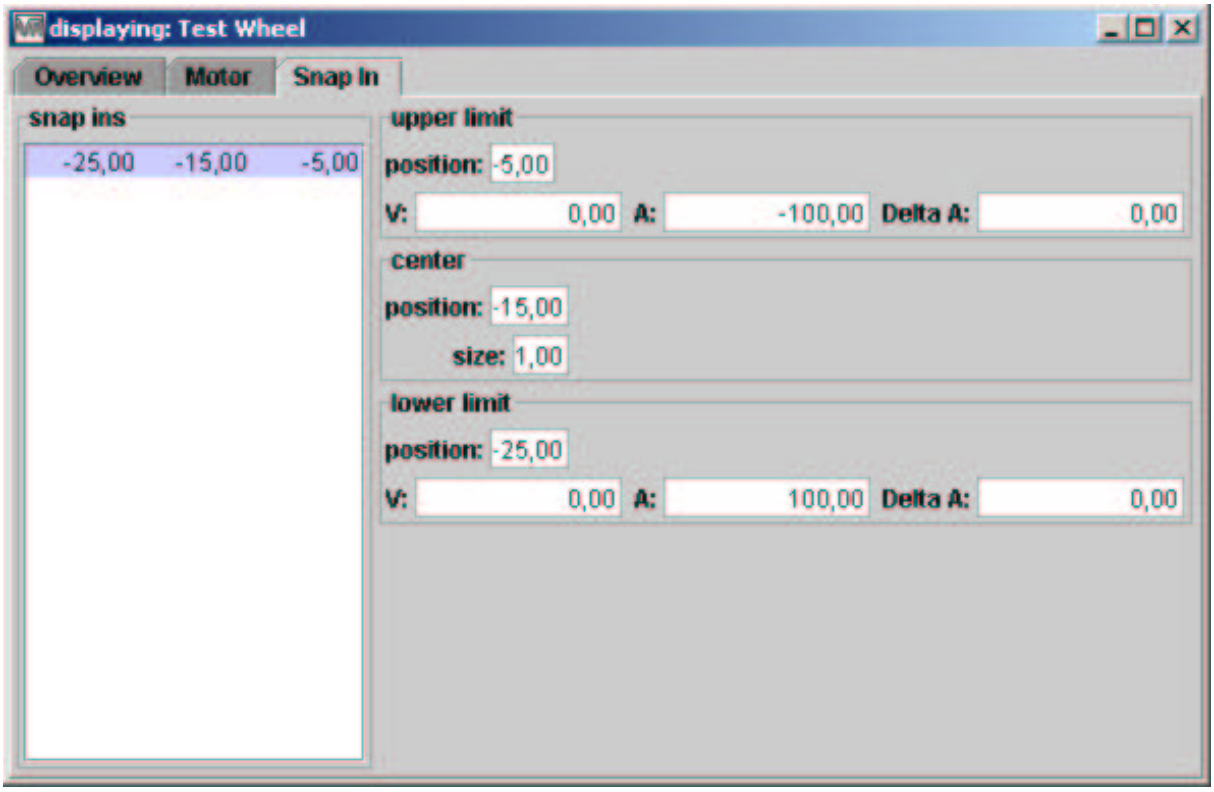


Abbildung 5.4: Rastendarstellung

das ausgewählte Fahrprofil benötigt werden durch einen weißen Hintergrund hervorgehoben.

Durch diese strukturierte Aufteilung der Daten wird es dem Benutzer ermöglicht, diese einfach zu betrachten. Dabei wurde darauf geachtet, nur die nötigen Werte anzuzeigen bzw. entsprechend zu visualisieren, um den Benutzer nicht mit Daten zu überfluten.

Rastendarstellung

Die letzte Ansicht, die über die Reiter auszuwählen ist, ist die Rastendarstellung. In ihr werden alle an einem Element vorhandenen Rasten aufgelistet. Dies geschieht in der Form der Abb. 5.4. In der Rastendarstellung werden auf der linken Seite alle dem Element zugeordneten Rasten in Form einer Liste angezeigt. Jede Raste wird in dieser Liste durch ihren Anfangswinkel, Mittelpunkt, sowie Endwinkel repräsentiert. Wird in der Liste mit der Maus eine Raste selektiert, so werden ihre gesamten Daten angezeigt.

Für die Anzeige der Rastendaten wird, wie in Abb. 5.4 zu sehen, eine gesonderte Anzeigefläche benutzt. In dieser Anzeigefläche werden der Anfangs- und Endwinkel auf die gleiche Weise visualisiert. Unter der Winkelposition des Rastenbeginns/Endes befinden sich die jeweiligen Beschleunigungsdaten[4], die für eine Raste benötigt werden. Wenn aufgrund der Rastenkonstruktion kein Anfangs- bzw. Endwinkel existiert, werden die entsprechenden Textfelder grau hinterlegt, um den Benutzer visuell in der Datenerfassung zu unterstützen. Neben diesen Daten werden in der detaillierteren Anzeigefläche das Zentrum einer Raste sowie ihre Breite angezeigt.

Nachdem die sehr ausführliche Kontextansicht beschrieben wurde, wird nun die geometrische Ansicht behandelt.

[4]die Beschleunigungsdaten einer Raste werden in Kapitel 3 ausführlich beschrieben

5.2.3 Geometrische Ansicht

Die geometrische Ansicht ist ein zentrales Element der Visualisierung des virtuellen astronomischen Instruments. Durch die dreidimensionale Darstellung werden dem Benutzer enorme Mengen von Daten gleichzeitig präsentiert, ohne dass er dies bewusst wahrnimmt[5]. So vermittelt die geometrische Ansicht die Position, Ausrichtung, Interaktion, Geschwindigkeit, Übersetzungsverhältnis sowie etwaiges Fehlverhalten der Elemente. In der geometrischen Ansicht stehen dem Benutzer somit nahezu alle Daten aus der Kontextansicht zur Verfügung. Hinzu kommt, dass diese Ansicht nicht auf das Anzeigen eines einzelnen Elements beschränkt ist sondern die Daten aller selektierten Elemente visualisiert.

Die Steuerung der geometrischen Ansicht erfolgt sehr intuitiv über die in Unterkapitel 5.2.2 beschriebene Maussteuerung. Dadurch wird es dem Benutzer ermöglicht, selbst die Szenendarstellung auszuwählen. Die Szenendarstellung wird außerdem dadurch beeinflusst, welche Elemente der Benutzer in der Strukturansicht für die Visualisierung selektiert. Dies ist ein großer Vorteil des virtuellen astronomischen Instruments gegenüber dem realen Lucifer Instrument, da das reale Instrument dem Benutzer den Einblick in sein Inneres verwehrt. Bei dem virtuellen Instrument können alle uninteressanten Bauelemente ausgeblendet werden. Um diesen Effekt bei dem reellen Instrument zu erreichen, würde eine Gruppe von Mechanikern erforderlich sein, die das Lucifer Instrument den Benutzervorgaben entsprechend auseinandernehmen bzw. zusammensetzten.

In Abb. 5.5 ist die bereits erwähnte Testeinheit dargestellt. Die dreidimensionalen Modelle wurden von Klaus Weisbauer erstellt, indem die Existierende genau vermessen und dementsprechend in einer technischen Zeichnung umgesetzt wurde. In der geometrischen Ansicht aus Abb. 5.5 erkennt man deutlich die Montageplatte mit ihren drei Beinen. Vorne links sieht man den unterhalb eines kleinen Zahnrades angebrachten Motor. Dieser Motor treibt das simulierte Element an. Über das mit dem Motor verbundene Zahnrad wird ein weiteres kleines Rad sowie die eigentliche Testeinheit über einen Zahnriemen angetrieben. An dem großen Rad, das die eigentliche Testeinheit darstellt, sind zwei außenliegende Zapfen sowie die oberseitig angebrachte Kalibrierungsmarkierung zu erkennen. Diese Elemente treten mit den entsprechenden um dieses Rad angebrachten Sensoren in Verbindung. Sie sind bei der realen Testeinheit dazu notwendig, diese zu kalibrieren und die maximalen und minimalen Positionen festzulegen. In der geometrischen Ansicht werden sie benötigt, diese exponierten Positionen darzustellen.

Aus dem Vergleich der Abb. 5.5 mit der Abb. 5.6 geht deutlich hervor, wie ein einfaches geometrisches Modell durch unsere kognitive Wahrnehmung mit der reellen Testeinheit gleichgesetzt wird. Abb. 5.6 zeigt eine Aufnahme der reellen Testeinheit, die, zwecks besserer Vergleichbarkeit, in das Umfeld der geometrische Ansicht hineinkopiert wurde.

Aus dieser Abbildung ist außerdem zu erkennen, welche Modellinformationen weggelassen wurden. Bei der virtuellen Testeinheit erfolgt eine Beschränkung auf die für die Visualisierung wichtige Details.

In Abb. 5.7 ist eine selektierte Ansicht der Testeinheit dargestellt. Alle Visualisierungselemente, bis auf die Montageplatte, die den Motor, die Endschalter und den Zahnriemen enthält, sind ausgeblendet. Damit wird ersichtlich, welche Visualisierungsmöglichkeiten sich dem Benutzer bieten. Durch die Selektion der einzelnen Elemente kann die Komplexität einer Szene deutlich reduziert werden. So ist es außerdem möglich, dass Benutzer, die mit dem technischen Aufbau des astronomischen Instruments nicht vertraut sind, sich diesen über die geometrische Ansicht selbst erschließen. Dadurch, dass der Benutzer die entsprechenden Elemente in der Strukturansicht selektiert, werden die entsprechenden Bezeichnungen mit den dazugehörigen Bauelementen verknüpft.

[5] vergleiche Kapitel 3 in Bezug auf Informationsaufnahme des Menschen

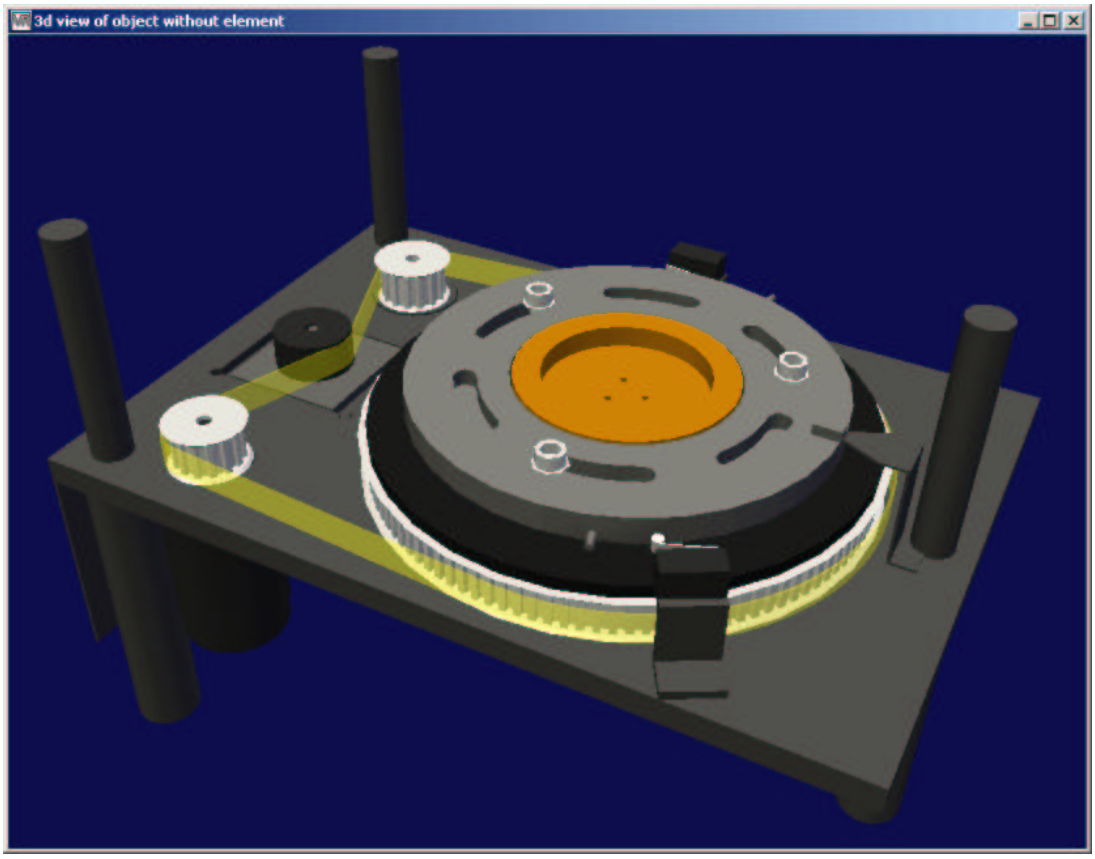


Abbildung 5.5: Geometrische Ansicht

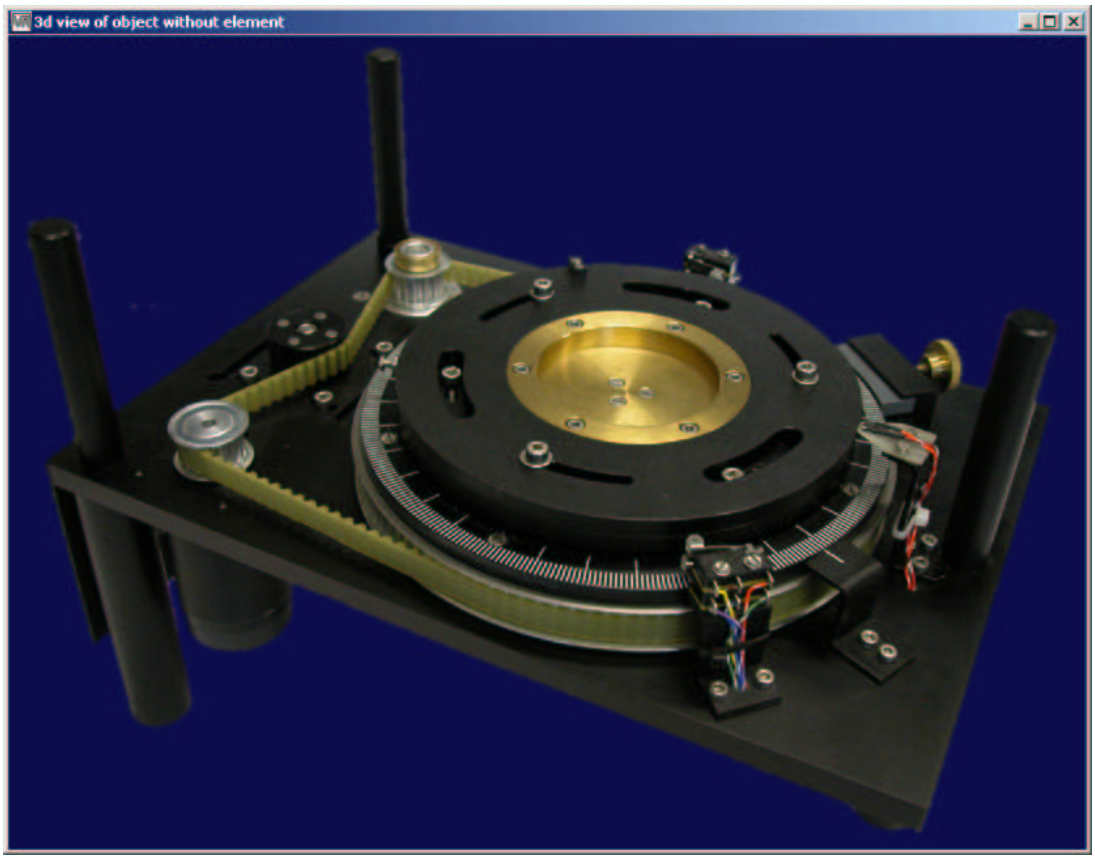


Abbildung 5.6: Vergleichsaufnahme der reellen Testeinheit

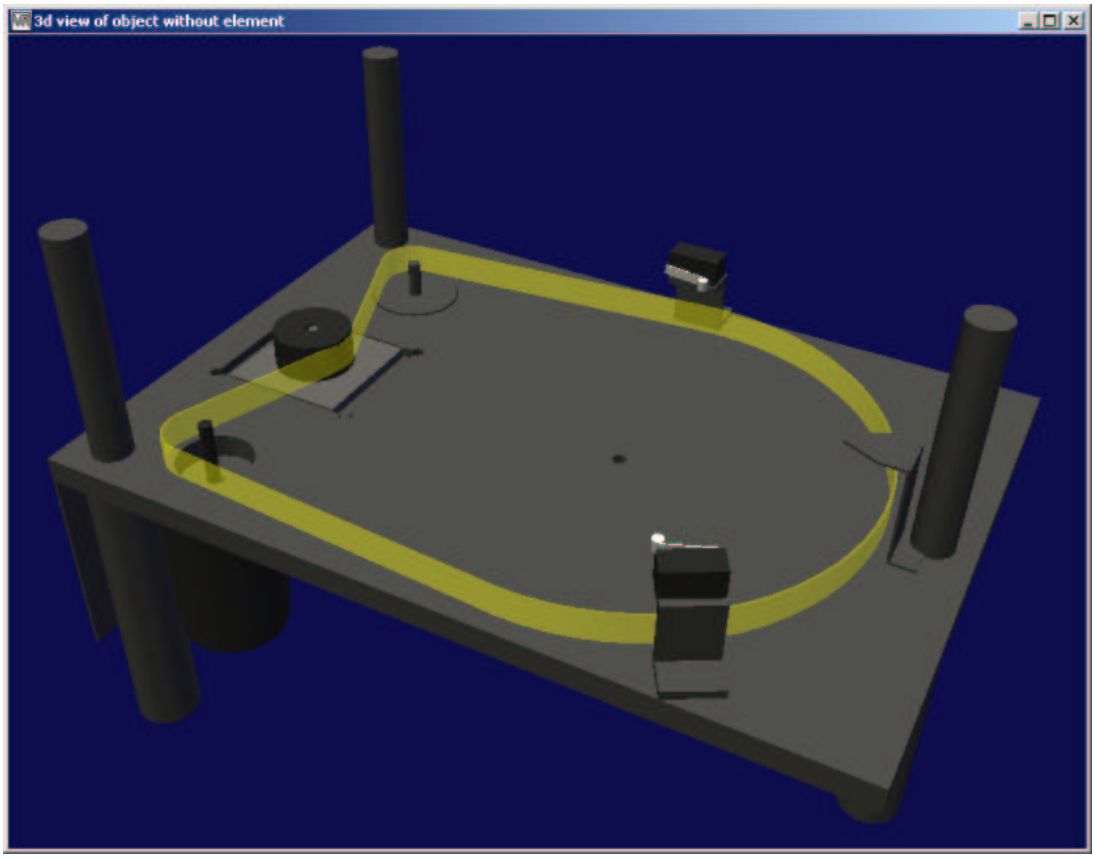


Abbildung 5.7: Geometrische Ansicht von ausgewählten Elementen

5.2.4 Kommunikationsdaten

Wird innerhalb des Hauptfensters aus Abb. 5.1 eine Netzwerkschnittstelle ausgewählt, erscheint ein Fenster, dessen Implementierung in Kapitel 4 beschrieben wurde. Das Aussehen dieses Fensters entspricht dem in Abb. 5.8 gezeigten. In der Titelleiste des Fensters werden die überwachte Netzwerkschnittstelle sowie der entsprechende Port angezeigt.

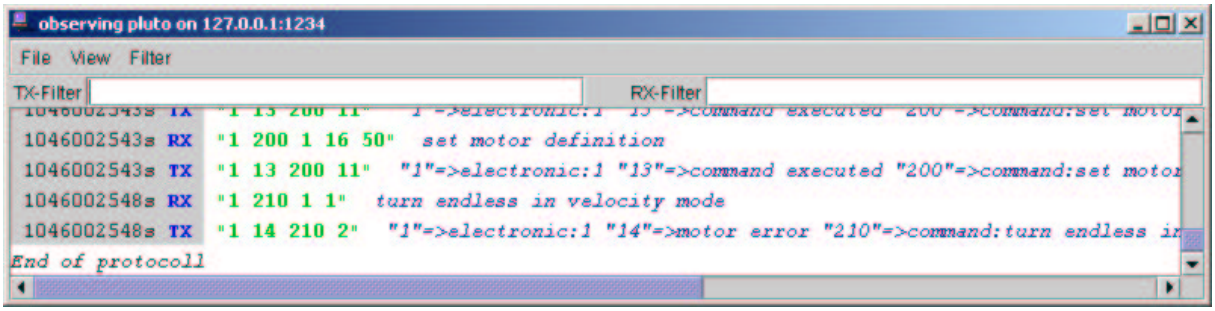


Abbildung 5.8: Darstellung der Kommunikationsdaten

Damit wird es dem Benutzer ermöglicht, auch bei mehreren überwachten Netzwerkadressen den Überblick zu behalten. Außerdem enthält die Titelleiste ein kleines Symbol, das eintreffende Daten durch Blinken signalisiert.

Wichtigster Bestandteil der Visualisierung der Kommunikationsdaten ist die textuelle Listendarstellung. In diesem Bereich werden die Kommunikationsdaten entsprechend textuell aufbereitet. In Abb. 5.8 ist ein beispielhafter Datenausschnitt zu sehen. Da die Kommunikationsdaten nach ihrem zeitlichen Eintreffen sortiert werden, befindet sich am Anfang jeder Zeile ein Zeitstempel. Auf diesen Zeitstempel folgt eine kleine Markierung, die aussagt, ob Daten

empfangen oder gesendet wurden. Das Kürzel *„TX“* steht für gesendete, das Kürzel *„RX“* für empfangene Daten. Darauf folgt der jeweilige Datensatz, der entsprechend seinem Status eingefärbt wird. So zeigt Grün an, dass die Daten verstanden wurden. Die gelbe Farbe signalisiert, dass die Daten zwar verstanden wurden, jedoch nicht der Spezifikation entsprechen. Dies kann dadurch passieren, dass die Elektronik des Max Planck Instituts selbst die Spezifikation nicht einhält und z.B. auf einen abgeschlossenen Befehl folgende Daten ignoriert anstatt eine Fehlermeldung zu erzeugen. Die rote Farbe wird dazu benutzt, Daten zu kennzeichnen, die von dem virtuellen Instrument nicht verstanden wurden und somit einen Fehler darstellen. Auf die Darstellung der Kommunikationsdaten folgt eine kurze Beschreibung dieser Daten, damit der in Zahlen kodierte Inhalt auch von anderen Benutzern verstanden werden kann.

Oberhalb der textuellen Datenvisualisierung befinden sich zwei Textfelder. In ihnen kann der Benutzer getrennt für die empfangenen und gesendeten Daten Filter vorgeben. Sind Filtertexte eingetragen, werden nur Datenpakete, die den entsprechenden Text enthalten, angezeigt.

Über die Menüleiste, die in Abb. 5.9 angezeigt wird, kann der Benutzer noch zusätzliche Aktionen ausführen.

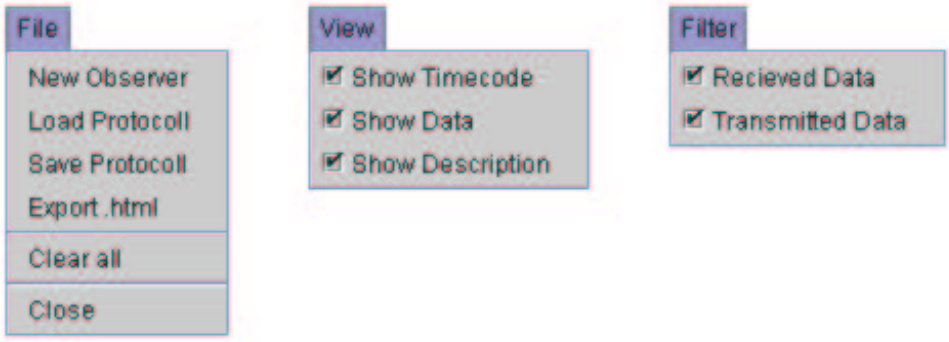


Abbildung 5.9: Menüstruktur der Kommunikationsansicht

Unter den Menüpunkt *„Filter“* kann der Benutzer bestimmen, welche Art von Kommunikationsdaten angezeigt werden sollen. Er hat die Möglichkeit, zwischen übertragenen und empfangenen Daten sowie jeder daraus machbaren Kombination zu wählen. Wenn den Benutzer z.B. nur die gesendeten Daten interessieren, hat er so die Option, die Empfangenen auszublenden.

Der Menüpunkt *„View“* ermöglicht, den Inhalt der Datenzeilen zu bestimmen. Hierüber kann er auswählen, ob der Zeitstempel, das Datenpaket oder der Beschreibungstext angezeigt werden soll. Diese Selektionsmöglichkeit dient der in Kapitel 3 genannten Fokussierung auf für den Benutzer wichtige Daten.

Mit dem Menüpunkt *„File“* werden dem Benutzer Funktionen zur Steuerung von Kommunikationsansichten gegeben. *„New Observer“* öffnet eine neue Kommunikationsansicht zu der in dieser Ansicht angezeigten Netzwerkschnittstelle. Die kann von dem Benutzer, in Verbindung mit den Selektionsmöglichkeiten, dazu genutzt werden, verschiedene gefilterte Ansichten einer Netzwerkschnittstelle darzustellen. So kann z.B. ein Fenster für das Anzeigen empfangener Daten und ein weiteres für das Anzeigen gesendeter benutzt werden. Der *„File“* Menüpunkt bietet außerdem die Möglichkeit, das Fenster zu schließen oder seinen Inhalt zu löschen. Wird einer der Dateidialoge zum Speichern, Laden oder Exportieren der Protokolldaten aufgerufen, erscheint ein der Abb. 5.10 entsprechendes Fenster. In diesem Fenster kann der Benutzer die zu speichernde / exportierende bzw. zu ladende Datei auswählen. Der Export der Protokolldaten erfolgt in dem gängigen HTML-Format und ist somit in jedem Webbrowser anzeigbar.

Nachdem die Benutzungsoberfläche näher beschrieben wurde, wird im folgenden Unterkapitel das virtuelle Instrument mit einem real existierenden verglichen. Da das Lucifer Instrument

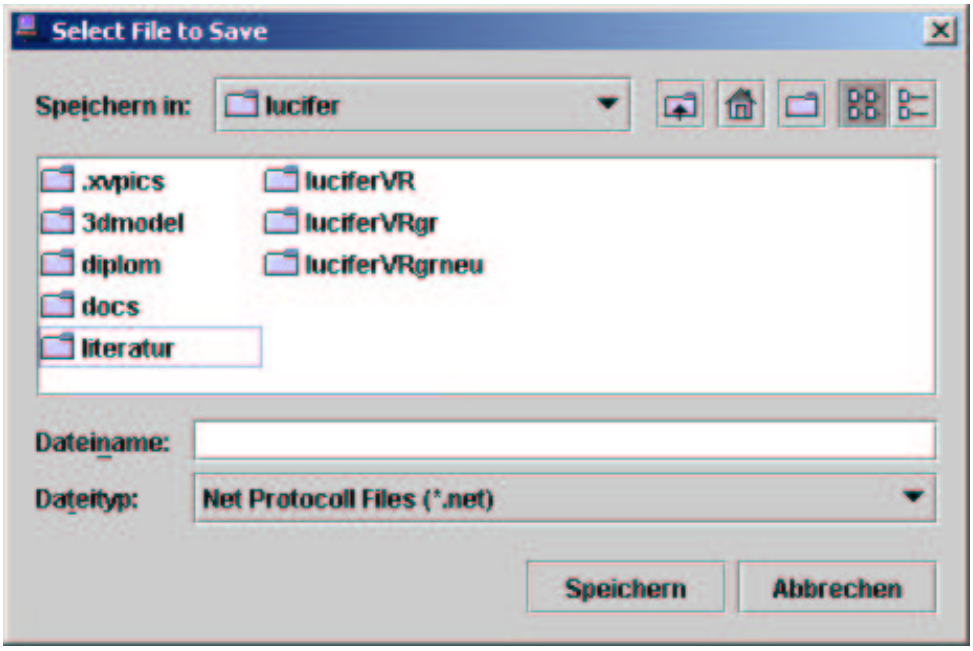


Abbildung 5.10: Dateidialog der Kommunikationsansicht

sich noch in der Planungphase befindet und somit nicht für einen Vergleich genutzt werden kann, wird eine real existierende Testeinheit zur Datengewinnung benutzt.

5.3 Ergebnisse

In diesem Unterkapitel wird die beschriebene Testeinheit mit ihrem virtuellen Gegenstück verglichen. Mittels einer *„Telnet“*-Verbindung werden die entsprechenden Steuerkommandos an sowohl die reale Elektronik als auch an das virtuelle Instrument geschickt. Diese Steuerkommandos sind Anhang C zu entnehmen. Die Eingabe von z.B. *„1 20“* veranlasst die Elektronik ihre Firmwareversion zurückzugeben.

Bei den Tests wurden alle bekannten Steuerkommandos an die beiden Elektroniken verschickt. Dabei reagieren beide, sowohl die Reale als auch die Virtuelle, gleich. Werden Fahrbefehle durch die Steuerkommandos ausgeführt, ist es nicht möglich, einen zeitlichen Unterschied bei den Fahrtzeiten festzustellen. Die nach Beendigung eines Fahrbefehls zurückgegebenen Werte des virtuellen Instruments entsprechen bis auf wenige Promille denen der realen Elektronik. Das virtuelle Instrument ist daher in einer Art Turing-Test[6] nicht von einem real existierenden zu unterscheiden. Diese Aussagen gelten nur für den Fall, dass weder die reelle Testeinheit noch das virtuelle Instrument eine Fehlerfunktion haben. Da das Fehlverhalten eines Motors physikalisch nicht genau zu beschreiben ist, weicht hier das Verhalten der realen Einheit von der virtuellen Einheit ab. Dieser Unterschied ist hierbei durch die Abweichung der Zufallsfunktion vom tatsächlichen Fehler gegeben.

Somit kann die Entwicklung der Steuersoftware für das Lucifer Instrument durch die neuen Testmöglichkeiten rechtzeitig zu Ende geführt werden.

In dem nächsten Kapitel folgt eine kurze Zusammenfassung, sowie ein Ausblick für das virtuelle astronomische Instrument.

[6]Der Mathematiker Alan Turing (1912-1954) stellte 1950 einen Test in Form eines Spiels vor, mit Hilfe dessen ein künstlicher Gesprächspartner von einem Realen unterschieden werden sollte. Konnte kein Unterschied festgetellt werden, hatte der künstliche Gesprächspartner den Turing-Test bestanden.

Wir werden nicht durch die Erinnerung an unsere Vergangenheit weise, sondern durch die Verantwortung für unsere Zukunft.

George Bernhard Shaw

Kapitel 6

Zusammenfassung und Ausblick

Ziel der Arbeit war es, Methoden zur Realisierung eines virtuellen astronomischen Instruments sowie einer entsprechenden Visualisierung zu entwickeln. Eine grundlegende Neuheit besteht darin, dass in der Astronomie zuvor noch nie ein virtuelles Instrument zu Testzwecken einer Steuersoftware entwickelt wurde. Für die Erstellung eines Softwareprototypen war es nötig, ein Simulationsmodell zu entwerfen, das das reale Zeitverhalten der Instrumentenmechanik nachempfindet. Diese Emulation musste außerdem mit einem entsprechenden Instrumentenmodell vereint werden, damit eine Visualisierung der Mechanik in einer dreidimensionalen Ansicht möglich wurde.

Das größte Problem bei der Entwicklung des virtuellen astronomischen Instruments war (und ist), dass sich das gesamte Instrument noch in der Entwicklungsphase befindet und somit Änderungen in Design und Umsetzung sehr häufig vorkommen. Um diesen Änderungen gerecht zu werden, musste der Softwareprototyp eine sehr flexible Programmierung aufweisen, damit er auf schnelle und einfache Art anzupassen ist.

Da sich das Lucifer Instrument noch in seiner Entwicklungsphase befindet, ist die Dokumentation der Steuerelektronik selbst noch nicht abgeschlossen und somit Grund dafür, dass das Verhalten der Elektronik nur durch umfangreiche Tests ermittelt werden konnte.

Neben einer auf die Steuerelektronik zugeschnittenen Motorensimulation mussten Methoden für die Kommandoerkennung entwickelt werden, da die Kommandos keiner festen Schematik folgen.

Damit der Softwareprototyp für die Entwicklung der Steuersoftware eingesetzt werden kann, ist es notwendig, die entsprechenden Instrumentendaten zu erhalten und somit das virtuelle Instrument fertigzustellen. In Zukunft kann der Softwareprototyp dazu benutzt werden, Instrumente, die über eine ähnliche Steuerelektronik verfügen, nachzubilden.

Durch das fertige virtuelle astronomische Instrument ergeben sich folgende Vorteile:

- Die Steuersoftware kann bereits vor Fertigstellung des realen Instruments getestet werden. Somit können im Vorfeld schon logische Fehler eliminiert werden, so dass die fertige Steuersoftware nur an das reale Instrument angepasst werden muss.
- Neue Softwareversionen können an dem virtuellen Instrument getestet werden, ohne teure Beobachtungszeit am realen Instrument zu benötigen.
- Astronomen können sich durch Verwendung des virtuellen Instruments mit der Steuerung bereits in ihrem Land vertraut machen, um sich so in den USA ganz ihrer wissenschaftlichen Arbeit zu widmen.

Im Rahmen dieser Diplomarbeit haben sich einige neue Aufgabenstellungen ergeben: Zum einen könnte die Visualisierung in eine interaktive Steuerung des virtuellen Instruments umgewandelt werden, damit eine Einstellung des Instrumentenzustandes einfacher möglich ist, zum anderen könnte die zur Zeit noch im Quelltext stattfindende Konfiguration in eine externe

Datei ausgelagert werden. Dazu wäre es empfehlenswert, eine eigene XML (*extensible markup language*) -Definition zu entwickeln und einen entsprechenden Parser zu implementieren. Ein weiterer für das virtuelle astronomische Instrument interessanter Aspekt wäre die Entwicklung von Möglichkeiten, aus einer bereits vorhandenen Datenbank die entsprechenden Himmelsausschnitte nach den durch die Instrumentenkonfiguration gegebenen Parametern zu verändern und auszugeben. Würde noch die Teleskopbewegung simuliert, wäre das virtuelle astronomische Instrument wie ein reales zu verwenden.

Der Softwareprototyp ist bereits den Mitarbeitern der Elektronikgruppe des Max Planck Instituts für Astronomie vorgestellt worden. Diese zeigten sich von den Möglichkeiten begeistert, die sich durch eine, auf ihre Elektronik angepasste, frei konfigurierbare, virtuelle Testumgebung bieten. Es ist beabsichtigt, den Softwareprototyp für weitere Projekte einzusetzen.

Anhang A

Glossar

In diesem Glossar werden die benutzte Abkürzungen sowie Fachbegriffe kurz erklärt.

AIP Astrophysikalisches Institut Potsdam.

AIRUB Astronomisches Institut der Ruhr-Universität Bochum.

API (application program interface) Schnittstelle zur Einbindung von extern bereitgestellten Funktionen.

Bytecode Vorübersetzte Form der Java Programme.

FHTG Die Fachhochschule für Technik und Gestaltung in Mannheim.

Firmware Software, die das Verhalten einer programmierbaren Elektronik bestimmt.

GMT (Greenwich mean time) Referenzzeitzone, die durch Greenwich (London) definiert wird.

GUI (graphical user interface) Graphische Benutzungsoberfläche. Wird zur Kommunikation zwischen Benutzer und Programm verwendet.

HTML (hypertext markup language) Strukturiertes und verweisendes Dokumentenformat, das im Internet benutzt wird.

Internet Weltumspannendes Kommunikationsnetz.

Java Objektorientierte Programmiersprache.

Java3d Softwarepaket, das dreidimensionale Graphik für Java ermöglicht.

Javadoc System zum Erstellen von Programmdokumentationen in Form von HTML.

JDK (java developers kit) Entwicklungsumgebung für Javaprogramme, die von der Firma Sun zur Verfügung gestellt wird.

JRE (java runtime environment) Virtuelle Maschine zur Ausführung des Bytecode.

Kryotechnik (kryo = kalt) Tiefsttemperaturtechnik zur Erzeugung von Temperaturen unter -100°C.

LBT (large binocular telescope) Großteleskop das auf dem Mount Graham nahe Tucson in Arizona errichtet wird.

LSW Landessternwarte Heidelberg.

LUCIFER (Lbt near infrared spectroscopic Utility with Camera and Integral Field unit for Extragalactic Research) Das nachgebildete astronomische Instrument.

MPE Max Planck Institut für extraterrestrische Physik in Garching.

MPIA Max Planck Institut für Astronomie in Heidelberg.

MPIfR Max Planck Institut für Radioastronomie in Bonn.

OpenGL Standardisierte Schnittstelle für dreidimensionale Graphikprogrammierung.

Port Vom TCP / IP verwendete Netzwerkadressenunterteilung. Bis zu 65535 dieser Ports stellt das TCP/IP den Programmen zur Netzwerkkommunikation zur Verfügung.

RS232 Norm für serielle Kommunikation über 3 Leitungsadern.

TCP/IP (transmission control protocol / internet protocol) Protokoll zum Übertragen von Daten in Rechnernetzen.

Telnet Ein Programm für auf Zeichen basierende Kommunikation über Rechnernetzwerke.

Thread Programmteil, der innerhalb des gleichen virtuellen Speichers des Hauptprogramms ausgeführt wird und dabei seine zur Programmausführung wichtigen Daten selbst verwaltet.

UML (unified modelling language) Eine strukturierte Sprache zur Dokumentation von Prozessen.

Webbrowser Programm zum betrachten verknüpfter HTML Dokumente

XML (extensible markup language) Frei konfigurierbares Strukturierungsmodell für Textdaten.

Anhang B

Symbolindex

v : Geschwindigkeit
(in Schritte pro Sekunde)

v_0 : Anfangsgeschwindigkeit einer Bewegung
(in Schritte pro Sekunde)

v_{des} : Gewünschte Geschwindigkeit
(in Schritte pro Sekunde)

v_{acc} : Geschwindigkeitszuwachs während der Beschleunigungsphase
(in Schritte pro Sekunde)

v_{sim} : Maximal zulässige Geschwindigkeit innerhalb der Simulation
(in Schritte pro Sekunde)

t : Zeit
(in Sekunden)

t_{acc} : Zeit der beschleunigten Beschleunigungsphase
(in Sekunden)

t_{lin} : Zeit der linearen Beschleunigungsphase
(in Sekunden)

t_{rest} : Zeit der Plateauphase
(in Sekunden)

a : Beschleunigung
(in Schritte pro Sekunde^2)

σ : Vorzeichen der Beschleunigungsänderung

$\dot{a}$: Beschleunigungsänderung
(in Schritte pro Sekunde^3)

a_{des} : Gewünschte Beschleunigung
(in Schritte pro Sekunde^2)

a_{acc} : Nach Beschleunigungsphase erreichte Beschleunigung
(in Schritte pro Sekunde^2)

a_{sim} : Maximal zulässige Beschleunigung innerhalb der Simulation
(in Schritte pro Sekunde^2)

ω : Winkelausrichtung des Motors
(in Grad)

ω_0 : Startwinkel
(in Grad)

ω_{des} : Gewünschte Winkelveränderung
(in Grad)

Anhang C

Steuerkommandos

Die folgende Liste enthält alle bekannten Steuerkommandos. Aufgrund der sich noch in der Erstellung befindenden Dokumentation ist die Liste nur hinreichend genau. Die einzig feste Struktur, die bei jedem Befehl festzustellen ist, dass eine Kartenkennung dem Befehl vorzustellen ist. Bei der Verbindung mehrerer Steuerkarten und der Kommunikation über eine gemeinsame serielle Schnittstelle dient diese Kartenkennung der Adressierung. In dem Standardfall sollte eine Kartennummer von *„1“* gewählt werden.

Befehl	**Reset**
Syntax	*<Karten Nr.> <8>*
Beschreibung	Reinitialisiert die Elektronik
Rückmeldungen	unbekannt

Befehl	**Get Firmware Version**
Syntax	*<Karten Nr.> <20>*
Beschreibung	Abfrage der Firmwareversion
Rückmeldungen	**<Firmwareversion>**

Befehl	**Find Home**
Syntax	*<Karten Nr.> <217> <Motor Nr.>*
Beschreibung	Führt die Motorkalibrierung durch
<Motor Nr.>	{**1..8**}
Rückmeldungen	**<Karten Nr.> <12> <217> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <217> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <217> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <217> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <217> <4>** Kein Referenzschalter vorhanden **<Karten Nr.> <14> <217> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <1> <Pos>** Bewegung beendet mit Position <Pos>

Befehl	**Set Motor Definition**
Syntax	*<Karten Nr.> <200> <Motor Nr.> <Auswahl> <Wert>*
Beschreibung	Definition eines Motorkanals
<Motor Nr.>	{**1..8**}
<Auswahl>	{**1**} = Fahrprofil {**2**} = Geschwindigkeit {**3**} = Startgeschwindigkeit {**4**} = Beschleunigung {**5**} = Max. Beschleunigung {**6**} = Beschleunigungsänderung {**8**} = Mikroschrittweite {**9**} = Schritte pro Motor-Umdrehung {**10**} = Motor-Übersetzung {**15**} = Aufbaustruktur {**16**} = Geschwindigkeit für Kallibrierung
<Wert>	zu 1 {**1**}=S-Kurve; {**2**}=Trapezoid; {**3**}=ohne) zu 2 {**0..50000000**} steps/sec bzw. steps/cycle zu 3 {**0..50000000**} steps/sec bzw. steps/cycle zu 4 {**143..2147483647**} steps/sec^2 bzw. steps/cycle^2 zu 5 {**143..4656612**} steps/sec^2 bzw. steps/cycle^2 zu 6 {**7..2147483647**} steps/sec^3 bzw. steps/cycle^3 zu 8 {**1, 2, 4, 8, 16**} zu 9 {**0..65000**} zu 10 {**0..65000**} zu 15 {**0x0..0xFFFF**} Bit 0 = Existenz positiver Endschalter Bit 1 = Existenz negativer Endschalter Bit 2 = Kalibrierungssonde liegt auf positivem Endschalter Bit 3 = Kalibrierungssonde liegt zwischen Endschaltern Bit 4 = Kalibrierungssonde liegt auf negativem Endschalter Bit 5 = Anfahrtsrichtung Kalibrierung Bit 12 = Nach Bewegung Motor Stromlos Bit 13 = Gefahrene Strecke bei Kalibrierung ausgeben Bit 14 = Meldung nach Bewegungsende zu 16 {**0..50000000**} steps/sec bzw. steps/cycle
Rückmeldungen	**<Karten Nr.> <12> <200> <n>** n-ter Parameter ist falsch **<Karten Nr.> <13> <200> <Motor Nr.><0>** Befehl erfolgreich / weitere Parameter benötigt **<Karten Nr.> <13> <200> <Motor Nr.><1>** Befehl erfolgreich / alle nötigen Parameter gesetzt **<Karten Nr.> <14> <200> <5>** Motor bewegt sich und kann nicht verändert werden

Befehl	**Get Motor Definition**
Syntax	*<Karten Nr.> <201> <Motor Nr.> <Auswahl>*
Beschreibung	Abfrage der Definition eines Motorkanals
<Motor Nr.>	{**1..8**}
<Auswahl>	{**0**} = Alles ausgeben {**1**} = Fahrprofil {**2**} = Geschwindigkeit {**3**} = Startgeschwindigkeit {**4**} = Beschleunigung {**5**} = Max. Beschleunigung {**6**} = Beschleunigungsänderung {**8**} = Mikroschrittweite {**9**} = Schritte pro Motor-Umdrehung {**10**} = Motor-Übersetzung {**15**} = Aufbaustruktur {**16**} = Geschwindigkeit für Kallibrierung
Rückmeldungen	**<Karten Nr.> <12> <201> <n>** n-ter Parameter ist falsch **<Karten Nr.> <201> <Motor Nr.> <Ergebnis>** Gibt den angefragten Wert, in der in den Optionen festgelegten Formatierung, aus

Befehl	**Set Option**
Syntax	*<Karten Nr.> <202> <Auswahl> <Wert>*
Beschreibung	Setzt die Elektronikeinstellungen
<Auswahl>	{**0**} = Komplette Eingabe {**1**} = Positionswerte nach Fahrbewegung {**2**} = Einheit der Eingaben {**3**} = Ausgleich der Mikroschritte bei Positionsausgabe {**4**} = Ausgleich der Mirkoschritte bei Bewegung **{5}** = Aktivierung der Textausgabe
<Wert>	zu 0 {**0x0..0x1F**} Eingabe Binärkodiert zu 1,3,4,5 {**0**} = Aus, {**1**} = An zu 2 {**0**} = cycle, {**1**} = second
Rückmeldungen	**<Karten Nr.> <12> <202> <n>** n-ter Parameter ist falsch

Befehl	**Get Option**
Syntax	*<Karten Nr.> <203>*
Beschreibung	Abfrage der Elektronikeinstellungen
Rückmeldungen	**<Karten Nr.> <12> <203> <n>** n-ter Parameter ist falsch
	<Karten Nr.> <203> <Ergebnis>
	Gibt den angefragten Wert, in der in den Optionen festgelegten Formatierung hexadezimal oder textuel, aus

Befehl	**Get Register Info**
Syntax	*<Karten Nr.> <205> <Motor Nr.> <Auswahl>*
Beschreibung	Abfrage der Motorchipsatz-Register
<Motor Nr.>	{**1..8**}
<Auswahl>	{**0**} = Alles ausgeben {**1**} = Position {**5**} = Geschwindigkeit
Rückmeldungen	**<Karten Nr.> <12> <205> <n>** n-ter Parameter ist falsch **<Karten Nr.> <205> <Motor Nr.> <Ergebnis>** Gibt den angefragten Wert, in der in den Optionen festgelegten Formatierung, aus

Befehl	**Go Velocity**
Syntax	*<Karten Nr.> <210> <Motor Nr.> <Richtung>*
Beschreibung	Fährt den angegebenen Motor ohne Fahrprofil beliebig lange
<Motor Nr.>	{**1..8**}
<Richtung>	{**0**} = negative Richtung {**1**} = positive Richtung
Rückmeldungen	**<Karten Nr.> <12> <210> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <210> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <210> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <210> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <210> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Positiver Endschalter mit Position <Pos> erreicht **<Karten Nr.> <16> <Motor Nr.> <3> <Pos>** Negativer Endschalter mit Position <Pos> erreicht

Befehl	**Motor Stop**
Syntax	*<Karten Nr.> <211> <Motor Nr.>*
Beschreibung	Stoppt den angegebenen Motor
<Motor Nr.>	{**1..8**}
Rückmeldungen	**<Karten Nr.> <12> <211> <n>** n-ter Parameter ist falsch

Befehl	**Motor Smooth Stop**
Syntax	*<Karten Nr.> <212> <Motor Nr.>*
Beschreibung	Stoppt den angegebenen Motor mit einer Bremsphase
<Motor Nr.>	{**1..8**}
Rückmeldungen	**<Karten Nr.> <12> <212> <n>** n-ter Parameter ist falsch

Befehl	**Go n Steps**
Syntax	*<Karten Nr.> <213> <Motor Nr.> <Schritte>*
Beschreibung	Fährt den angegebenen Motor die angegebene Anzahl an Schritten
<Motor Nr.>	{**1..8**}
<Schritte>	{**-2147483647..2147483647**}
Rückmeldungen	**<Karten Nr.> <12> <213> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <213> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <213> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <213> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <213> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <1> <Pos>** Bewegung beendet mit Position <Pos> **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Positiver Endschalter mit Position <Pos> erreicht **<Karten Nr.> <16> <Motor Nr.> <3> <Pos>** Negativer Endschalter mit Position <Pos> erreicht

Befehl	**Go n Abs. Steps**
Syntax	*<Karten Nr.> <214> <Motor Nr.> <Schritte>*
Beschreibung	Fährt den angegebenen Motor an die in Schritten angegebene Position bezüglich des Nullpunktes
<Motor Nr.>	{**1..8**}
<Schritte>	{**-2147483647..2147483647**}
Rückmeldungen	**<Karten Nr.> <12> <214> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <214> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <214> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <214> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <214> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <1> <Pos>** Bewegung beendet mit Position <Pos> **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Positiver Endschalter mit Position <Pos> erreicht **<Karten Nr.> <16> <Motor Nr.> <3> <Pos>** Negativer Endschalter mit Position <Pos> erreicht

Befehl	**Go n Degrees**
Syntax	*<Karten Nr.> <215> <Motor Nr.> <Grad>*
Beschreibung	Fährt den angegebenen Motor die angegebene Gradzahl
<Motor Nr.>	{**1..8**}
<Grad>	{**-36000,0..36000,0**}
Rückmeldungen	**<Karten Nr.> <12> <215> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <215> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <215> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <215> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <215> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <1> <Pos>** Bewegung beendet mit Position <Pos> **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Positiver Endschalter mit Position <Pos> erreicht **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Negativer Endschalter mit Position <Pos> erreicht

Befehl	**Go n Abs. Degrees**
Syntax	*<Karten Nr.> <216> <Motor Nr.> <Grad>*
Beschreibung	Fährt den angegebenen Motor an die in Grad angegebene Position bezüglich des Nullpunktes
<Motor Nr.>	{**1..8**}
<Grad>	{**-36000,0..36000,0**}
Rückmeldungen	**<Karten Nr.> <12> <216> <n>** n-ter Parameter ist falsch **<Karten Nr.> <14> <216> <1>** Motor ist nicht vollständig initialisiert **<Karten Nr.> <14> <216> <2>** Motor befindet sich am positiven Endschalter **<Karten Nr.> <14> <216> <3>** Motor befindet sich am negativen Endschalter **<Karten Nr.> <14> <216> <5>** Motor bewegt sich und kann nicht verändert werden **<Karten Nr.> <16> <Motor Nr.> <1> <Pos>** Bewegung beendet mit Position <Pos> **<Karten Nr.> <16> <Motor Nr.> <2> <Pos>** Positiver Endschalter mit Position <Pos> erreicht **<Karten Nr.> <16> <Motor Nr.> <3> <Pos>** Negativer Endschalter mit Position <Pos> erreicht

Anhang D

Danksagungen

Ich möchte meinen Eltern dafür danken, dass sie mich mein Leben lang unterstützt haben und meine Fähigkeiten gefördert haben. Meiner Mutter danke ich besonders dafür, dass sie trotz des frühen Todes meines Vaters mir ein Studium finanziell ermöglicht hat.

Außerdem danke ich folgenden Menschen für die Ermöglichung meiner Diplomarbeit (in alphabetische Reihenfolge):

- Ralf-Jürgen Dettmar
  Für die Ermöglichung einer Diplomarbeit im astronomischen Umfeld, sowie seine Betreuung.

- Lutz Haberzettel
  Dafür, dass er als mein Zimmergenosse immer für Fragen zur Verfügung stand.

- Günter Hoppe
  Für die langjährige astronomische Freundschaft, durch die ich den Grundlagen der Astronomie näher gekommen bin, sowie der Hilfe beim Ausdrucken dieser Diplomarbeit.

- Markus Jütte
  Für die sehr gute Betreuung während meiner Diplomarbeit, sowie den Anmerkungen zu diesem Text.

- Michael Lemitz
  Für die tatkräftige Unterstützung im Umgang mit der Steuerelektronik des Max Planck Institutes.

- Eva Manthey
  Für Korrekturanregungen sowie die gelegentliche Nutzung ihres Computers.

- Claudio Moraga
  Für die Ermöglichung einer fächerübergreifenden Diplomarbeit und die Kooperation des LS1 in Dortmund mit dem astronomischen Institut der Ruhr-Universität Bochum.

- Ferdinand und Ethel Polsterer sowie meiner Großmutter Gisela Polsterer
  Für die finanzielle Unterstützung während meines Studiums, ohne die so manches in der Diplomarbeit verwandte Buch nicht mein Eigentum wäre.

- Daniela Quetting
  Dafür, dass sie mir während der Diplomarbeit den Rücken freigehalten hat.

- Wolfhard Schlosser
  Für die Hilfe bei mathematischen und technischen Problemen aller Art.

- Klaus Weisbauer
  Für die Unterstützung bei der Erstellung eines dreidimensionalen Modells der Testeinheit.

Allen nicht namentlich genannten, möchte ich auch meinen Dank für ihre Unterstützung ausdrücken.

Literaturverzeichnis

[Alh98] Sinan Si Alhir. *UML in a Nutshell*. O'Reilly + Associates, Inc., 1998.

[Bon03] Erich Bonnert. Auf dem chip-gipfel. *c't Magazin für Computer Technik*, (5), 2003.

[BSMM99] I.N. Bronstein, K.A. Semendjajew, G. Musiol, and H. Muehlig. *Taschenbuch der Mathematik*. Harri Deutsch, 4 edition, 1999.

[CL99] C. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Kluwer, 1999.

[Com01] Dougles E. Comer. *Computernetzwerke und Internets*. Pearson Education Deutschland GmbH, 3 edition, 2001.

[Dar01] Ian F. Darwin. *Java Cookbook*. O'Reilly + Associates, Inc., 2001.

[DD96] Doberkat and Dißmann. *Einführung in die objektorientierte Programmierung mit BETA*. Addison Wesley Longman, Inc., 1996.

[Fla98] David Flanagan. *Java in a Nutshell*. O'Reilly + Associates, Inc., 2 edition, 1998.

[Fla99] David Flanagan. *Java Foundation Classes in a Nutshell*. O'Reilly + Associates, Inc., 1999.

[Güt92] Ralf Hartmut Güting. *Datenstrukturen und Algorithmen*. Teubner, 1992.

[Hau00] Roland Hausser. *Grundlagen der Computerlinguistik*. Springer-Verlag, 2000.

[Ins82] Bibliographisches Institut. *Duden, Das Fremdwörterbuch*. F.A. Brockhaus AG, 4 edition, 1982.

[JRV+89] J. Johnson, W. Roberts, W. Verlank, D.C. Smith, C.H. Irby, M. Beard, and K. Mackey. The xerox star: A retrospective. *IEEE Computer*, (22(9)):11–29, 1989.

[Knu97] Donald Ervin Knuth. *The Art of Computer Programming*, volume 1 Fundamental Algorithms. Addison-Wesley, 3 edition, 1997.

[Knu98a] Donald Ervin Knuth. *The Art of Computer Programming*, volume 2 Seminumerical Algorithms. Addison-Wesley, 3 edition, 1998.

[Knu98b] Donald Ervin Knuth. *The Art of Computer Programming*, volume 3 Sorting and Searching. Addison-Wesley, 2 edition, 1998.

[Knu99] Jonathan Knudsen. *Java 2D Graphics*. O'Reilly + Associates, Inc., 1 edition, 1999.

[Kop00] Helmut Kopka. *LaTeX*, volume 1 Einführung. Addison-Wesley, 3 edition, 2000.

[Kot74] Ernst Kottke. *Schrittmotoren und ihre Anwendung*. Verlag Johanna Kottke, 1974.

[LK99] A.M. Law and W.D. Kelton. *Simulation modeling and analysis*. McGraw-Hill, 3 edition, 1999.

[LW98] Marc Loy and Dave Wood. *Java Swing*. O'Reilly + Associates, Inc., 1998.

[Mea00] Holger Mandel et al., editor. *LUCIFER: a NIR spectrograph and imager for the LBT*, 4008, Munich, March 2000. S.P.I.E. Companion Proceedings.

[MG83] Menge-Güthling. *Langenscheidts großes Schulwörterbuch, Lateinisch-Deutsch*. Langenscheidt KG, Berlin und München, 1983.

[Oea00] P. Osmer et al., editor. *Multiobject double spectrograph for the Large Binocular Telescope*, 4008, Munich, March 2000. S.P.I.E. Companion Proceedings.

[Per00] William E. Perry. *Effective Methods for Software Testing*. John Wiley + Sons, Inc., 2 edition, 2000.

[Rub98] R.Y. Rubinstein. *Modern simulation and modeling*. Wiley, 1998.

[SRD00] Henry Sowizra, Kevin Rushforth, and Michael Deering. *The Java 3D API Specification*. Addison-Wesley Inc., 2 edition, 2000.

[Str98] Thomas Strothotte. *Computational Visualization*. Springer Verlag, Berlin Heidelberg New York, 1998.

[Weg93] Ingo Wegener. *Theoretische Informatik*. B.G. Teubner, Stuttgart, 1993.

[Wie92] A.W. Wieder. Systems on chip: Die herausforderung der nächsten 20 jahre. *Informationstechnik it*, 4 92.

Abbildungsverzeichnis

1.1 Skizze des LBT (Quelle: EIE) . . . 8
1.2 LBT Gebäude . . . 9
1.3 Lucifer Instrument (Quelle: LSW) . . . 10

2.1 Steuerelektronik . . . 14
2.2 Funktionsweise eines Schrittmotors . . . 15
2.3 Verschiedene Fahrprofile . . . 16
2.4 Rastmechanismus . . . 18

3.1 Zerlegtes Fahrprofil . . . 22
3.2 Simulationsschema . . . 23
3.3 Beispiel Kommandostruktur . . . 29
3.4 GUI Layout für geometrische Modelle (Quelle: [Str98]) . . . 31

4.1 Softwarepakete des virtuellen Instrumentes . . . 35
4.2 Startbildschirm von Together . . . 40
4.3 Benutzungsoberfläche von Together . . . 41
4.4 Dokumentation im Webbrowser . . . 42
4.5 Datenstruktur der Simulation . . . 43
4.6 Klassen der Zeit- und Zufallserzeugung . . . 45
4.7 Bewegungsereignis . . . 46
4.8 Klassendiagramm der Simulation . . . 47
4.9 Schnittstelle zwischen Simulation und Elektronik . . . 48
4.10 Verwendung der Nachrichten . . . 50
4.11 Netzwerkanbindung . . . 51
4.12 Nachrichtenausgabe über die Netzwerkverbindung . . . 52
4.13 Nachbildung der Firmware . . . 53
4.14 Kommandostruktur . . . 54
4.15 Kommandoparser . . . 55
4.16 Visualisierung der Kommunikationsdaten . . . 56
4.17 Visualisierungsansichten der Mechanik . . . 57
4.18 Einfache Java3d Szenenstruktur . . . 58
4.19 Datenstruktur der Visualisierung . . . 59
4.20 Baumartige Elementstruktur . . . 59
4.21 Benutzte Szenenstruktur . . . 60
4.22 Übersicht der Klassen der Kontextansicht . . . 62
4.23 Visualisierung der Elementdaten . . . 63
4.24 Klassen zur Visualisierung der Motordaten . . . 65
4.25 Funktionsweise des Fahrtenschreibers . . . 66
4.26 Visualisierung der Rasten . . . 67

5.1 Hauptfenster mit Struktur- und Kommunikationsansicht 70
5.2 Elementdaten 72
5.3 Motordaten 73
5.4 Rastendarstellung 74
5.5 Geometrische Ansicht 76
5.6 Vergleichsaufnahme der reellen Testeinheit 76
5.7 Geometrische Ansicht von ausgewählten Elementen 77
5.8 Darstellung der Kommunikationsdaten 77
5.9 Menüstruktur der Kommunikationsansicht 78
5.10 Dateidialog der Kommunikationsansicht 79

Erklärung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit mit dem Thema

„Entwicklung und Visualisierung eines virtuellen astronomischen Instruments“

selbstständig und nur unter Verwendung der angegebenen Hilfsmittel erstellt habe. Weiterhin versichere ich, dass alle wörtlichen Zitate und die aus fremden Quellen übernommenen Informationen als solche von mir in Form von Literaturverweisen kenntlich gemacht wurden.

Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Dortmund, den 14.03.2003

Erklärung

Alle im Rahmen dieser Diplomarbeit entwickelten Programm-Quelltexte sind den Betreuern Prof. Dr.-Ing. Claudio Moraga und Prof. Dr. rer. nat. Ralf Jürgen Dettmar auf jeweils einer CD-Rom ausgehändigt worden.

Dortmund, den 14.03.2003