Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Björn Kaiser und André Gratz

Konzeption eines autonomen Zeppelins mit Patternerkennung und prototypischer Umsetzung

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Björn Kaiser und André Gratz

Konzeption eines autonomen Zeppelins mit Patternerkennung und prototypischer Umsetzung

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Gunter Klemke
Zweitgutachter : Prof. Dr. rer. nat. Kai von Luck

Abgegeben am 23. August 2007

Björn Kaiser und André Gratz

Thema der Bachelorarbeit

Konzeption eines autonomen Zeppelins mit Patternerkennung und prototypischer Umsetzung

Stichworte

Agentensysteme, UAV, Autonome Flugobjekte, Augmented Reality, Sensorik und Steuerung von Flugobjekten im Hochschulumfeld, Pattern-Erkennung.

Kurzzusammenfassung

Diese Arbeit umfasst die Konzeption eines autonom fliegenden Zeppelins, der mit Hilfe von Sensoren und Aktoren, insbesondere durch den Einsatz einer Kamerasteuerung und Techniken aus dem Bereich der „Augmented Reality (Patternerkennung)“ sich in einer ihm unbekannten Umgebung orientieren soll. Die Umsetzung erfolgt prototypisch auf einer neu entwickelten Roboterplattform.

Björn Kaiser und André Gratz

Title of the paper

Concept of an autonomous Zeppelin with a Pattern-Recognition-Control and prototyp Implementation

Keywords

Agentsystems, UAV, autonomous Flightobjects, Augmented Reality, Sensing and Controlling of Flightobjects in a University Ambient, Pattern-Recognition.

Abstract

This paper deals with the concept of a Zeppelin, which is to act autonomously in an unknown environment with help of sensors, actors and especially the use of Video Control and techniques from the subject of „Augmented Reality (Pattern Recognizing)“. The Implementation will take place on an prototyp driving robot.

Danksagung

Eine wissenschaftliche Arbeit wie diese, kann man schwerlich ohne Beistand und Hilfe bewältigen. Daher möchten wir auch die Personen hier erwähnen, die uns im Rahmen unserer Arbeit sehr geholfen haben.

Diese Arbeit umfasst mehrere wissenschaftliche Gebiete, wie z.B. Flugzeugbau, Schweißkunde, Werkkunde, Informatik, Robotik und Künstliche Intelligenz. Es war nötig mit vielen beteiligten Personen aus diesen einzelnen Fachgebieten zu interagieren und Wissen auszutauschen. Besondere Erwähnung sollen hier Herr Siemens aus dem Institut für Werkstoffkunde und Herr Svenson aus der Werkstatt der Hochschule für Angewandte Wissenschaften finden, die auch das Unmögliche haben entstehen lassen, damit der Traum eines Zeppelins und später der eines autonomen Roboters wahr wurden. Eine große Hilfe bei den Messungen und bei einigen Überlegungen zu der Konzeption war Burkhard Hübner, ein Kommilitone.

Unser Dank gilt ebenfalls unsere beiden Professoren Dr. rer. nat. Gunter Klemke und Dr. rer. nat. Kai von Luck, die fest an uns geglaubt haben, sogar im Urlaub mit uns im Kontakt standen und uns stets mit Anregungen versorgten.

Zu guter Letzt möchten wir uns bei unseren Familien bedanken, die auf vieles verzichten, auch mal Trost spenden mussten und stets für uns da waren.

Vielen Dank auch an alle, die hier nicht erwähnt wurden, aber trotzdem zum Gelingen dieser Bachelorarbeit nicht unwesentlich beigetragen haben.

Danke!

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Teil I.

Einführung

1. Einleitung

Agentensysteme bekommen eine immer größere Bedeutung in unserem Leben. Man findet sie überall wieder. Es gibt Agenten, die Lagerverwaltungsaufgaben übernehmen sowie Staubsauger, die vollautomatisch einen Raum reinigen. Was aber macht den Unterschied zwischen normalen Staubsaugern und solchen, die man Agenten nennen darf? Diese Frage ist nicht klar zu beantworten, doch kann man den Begriff Agent in Eigenschaften eingrenzen, die zumindest teilweise erfüllt sein müssen:

Autonom Kann eigenständig agieren

Reaktiv Reagiert rechtzeitig auf Änderungen in seiner Umgebung

Proaktiv Initiiert Aktionen, die seine Umgebung beeinflussen

Kommunikativ Kann Informationen mit Benutzern und anderen Agenten austauschen

Mobil Kann von einem System auf ein anderes migrieren

Adaptiv Lernfähig

Hierbei muss nicht jeder Agent alle Eigenschaften in sich vereinen. Es handelt sich eher um eine Richtlinie, was ein Agent alles leisten sollte bzw. könnte.

Wir werden uns im weiteren Verlauf dieser Arbeit auf die Betrachtung von autonomen Flugobjekten beschränken, die als Teil der Forschungsagenten wie folgt einzugliedern sind:

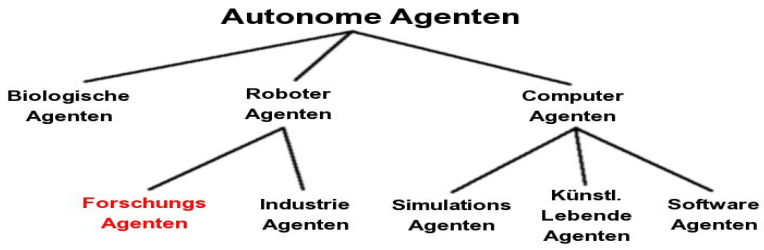


Abbildung 1.1.: Übersicht Agentensysteme

Autonome Flugobjekte sind demnach Agenten, die aus Hard- und Software bestehen. Sie besitzen zumindest begrenzte künstliche Intelligenz (Software), mit der die Eigenschaften

eines Agentensystems (teilweise) erfüllt werden. Außerdem besitzen autonome Flugobjekte eine Konstruktion und elektrische Komponenten, mit der sie ihre Umgebung wahrnehmen und beeinflussen können.

Sinn und Zweck solcher Agenten könnten unterschiedlicher nicht sein. Während die meisten, der heute hergestellten autonomen Flugobjekte für militärische Zwecke eingesetzt werden, ist der zivile und allgemeine Forschungszweck nicht minder wichtig. Denkbar sind in nicht allzu entfernter Zukunft fliegende Autos, die selbstständig zu Ihrem Ziel fliegen oder autonom fliegende Überwachungsroboter, die Verletzte aus Krisen- und Katastrophengebieten retten. Diese fliegenden Roboter können überall dort eingesetzt werden, wo Lebensgefahr droht.

Elektrische Komponenten und Computer werden mit der Zeit immer kleinere Ausmaße bei gleichbleibender oder sogar steigender Effizienz einnehmen. Dies wird völlig neue Anwendungsgebiete autonomer Flugobjekte mit sich bringen und es werden neue Schnittstellen zwischen der Maschine und dem Menschen definiert werden müssen.

Ziel dieser Bachelorarbeit ist es einen autonom fliegenden Zeppelin zu konzipieren. Insbesondere wird das Augenmerk auf die Machbarkeit und Realisierung von Flugobjekten im Umfeld einer Hochschule gerichtet. Die Umsetzung des Konzepts erfolgt auf einer ebenfalls autonom fahrbaren Plattform. Das Besondere hierbei ist die Ausstattung der Plattform mit einer Kamera, die dem Roboter die Möglichkeit bieten soll, sich anhand von Pattern zu orientieren. Der programmiertechnische Anteil unserer Arbeit wird in der speziellen Kameraerkennung aus dem Bereich der Augmented Reality realisiert.

Diese Arbeit ist in sechs Bereiche aufgeteilt:

Der erste Bereich, die Einführung, beinhaltet eine kurze Einleitung in die Thematik, sowie in Kapitel 2 einen Überblick über den Entwicklungsstand einiger Projekte, die dem hier erarbeiteten ähneln. Diese Beispiele kommen sowohl aus der Industrie wie auch aus verschiedenen Institutionen und Universitäten.

Der zweite Bereich beschäftigt sich mit der Analyse und den Anforderungen, die dem Bau eines Zeppelins für verschiedene Umgebungen zugrunde liegen. Hierbei wird insbesondere auf die Bewegung von Objekten in mehrdimensionalen Räumen eingegangen sowie der Möglichkeiten und Einschränkungen in der Konstruktion eines Zeppelins im universitären Umfeld. Der Bereich schließt mit dem Konzept eines Zeppelins und der möglichen Umsetzung.

Der dritte Bereich, das Design, gibt einen Einblick in das ARToolkit sowie in dessen Funktionsweise. Zusätzlich werden in diesem Bereich die Probleme erörtert die sich beim Bau des Zeppelins ergeben haben.

Im vierten Bereich, der sich mit der Realisierung des Projektes befasst, erklären wir den Aufbau und die Funktionen der Ersatzplattform. Hierbei wird insbesondere auf die Besonderheiten eingegangen, die durch die Portierung entstanden sind. Außerdem wird die eingesetzte Software detailliert erklärt.

Der fünfte Bereich ist eine Zusammenfassung des Erreichten, unserer Erkenntnisse und eines Ausblicks, bei dem auch auf die Möglichkeiten eingegangen wird, die durch diese Arbeit entstanden sind.

Im letzten Bereich, dem Anhang, befinden sich die Quellcodes der entwickelten Programme sowie deren Installation und Handhabung, die Beispielpattern aus dem ARToolkit und die Kostruktionspläne des Zeppelins.

2. Historisches: Von unbemannten Flugzeugen zu heutigen Drohnen

Große Verluste von Menschenleben während der Kriege ließen Wissenschaftler über die Möglichkeiten unbemannter Flugzeuge nachdenken. Dies war der erste Schritt in Richtung autonome Flugobjekte. Es wurden verschiedene Flugzeugmodelle entwickelt, die ferngesteuert bedient wurden.

Die Idee der autonomen Flugobjekte existiert bereits seit sehr langer Zeit. Diese wurde lange als geheim eingestuft, weshalb über deren Existenz bis vor 1980 wenig bekannt wurde. Die Forschung auf diesem Gebiet ist zu 98% militärischen oder Regierungszwecken vorbehalten und wird aus diesen Gründen geheim gehalten. Bekannt ist, dass es heutzutage etwas über 50 Hersteller mit über 150 verschiedenen Flugobjekten weltweit auf diesem Gebiet gibt.

2.1. Bomber B-17

Zwischen 1939 und 1945, während des Zweiten Weltkrieges, waren die ersten unbemannten Flugzeuge bereits im Einsatz. Nur wenige Leute wissen heutzutage, dass es sich hierbei um einige präparierte Exemplare des Bombers B-17 handelte.

Abbildung 2.1.: Bomber B-17 von der Boeing Airplane Company

Dieses besondere Flugzeug erbaut von der „Boeing Airplane Company“konnte bei einer Marsch- Geschwindigkeit von 293 km/h und einer maximalen Höhe von 10850 m eine Reichweite von 3219 km fliegen. Über die Ausstattung als unbemanntes Flugzeug ist nicht viel bekannt. Es muss sich hierbei allerdings um ein ferngesteuertes Flugzeug gehandelt haben ohne eine nennenswerte eigene künstliche Intelligenz. Unter Umständen wurde der Flug mit Hilfe einer Bordkamera überwacht und durch geschultes Personal am Boden kontrolliert. Das Flugzeug galt als extrem robust, da es im Notfall sogar mit nur einem der vier Neunzylinder Sternmotoren geflogen werden konnte.

2.2. Die V2 Rakete

Im Jahr 1936 wurde im Entwicklungswerk der Heeresversuchsanstalt Peenemünde (Deutschland) die A4 (Aggregat 4) entwickelt, die später zu Propagandazwecken während des Hitlerregimes in V2 (Vergeltungswaffe 2) umbenannt wurde. Hierbei handelte es sich um eine Rakete, die von A nach B fliegen konnte und dort angekommen, detonierte.

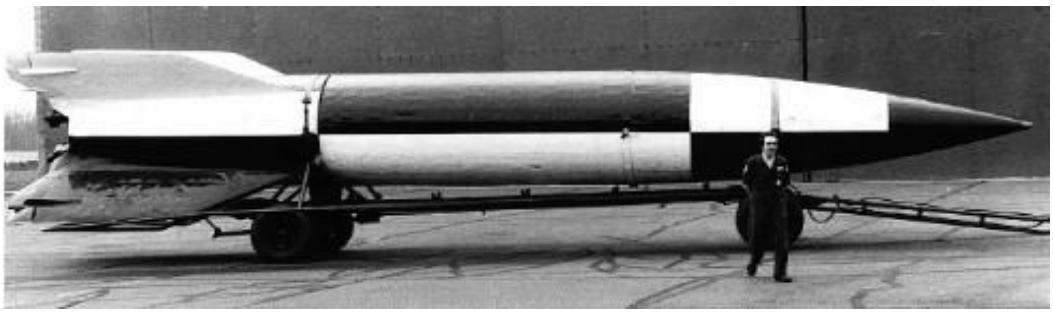


Abbildung 2.2.: V2 Prototyp

Bei der V2 handelte es sich konkret um eine unbemannte, (fern-)gesteuerte, ballistische Missile. Die Steuerung erfolgte über ein gyroskopisches System, dass im oberen Drittel der Rakete eingebaut war. In der Spitze selbst befand sich der Sprengstoff. Im mittleren Teil der Rakete befanden sich der flüssige Treibstoff und am unteren Ende die Antriebsturbine. Die Rakete besaß beim Start eine Schubkraft von 24947 N. Die Brennzeit des Treibstoffes betrug ca. 60 Sekunden und die Rakete erreichte dabei eine Geschwindigkeit von 1341 m/sek. Sie flog bis zu einer Höhe von 93 km und schaffte dabei eine Distanz von 361 km zu überwinden.

2.3. Global Hawk

Der Global Hawk ist ein unbemanntes Flugobjekt der Extraklasse. Es ist so groß wie ein mittelgroßer Passagierjet (Flügelspannweite von 35 m - max. Startgewicht von 11,5 Tonnen). Die Drohne kann bei einer Dienstgipfelhöhe von 20000 m (doppelt so hoch wie bei einem Passagierflugzeug) und einer Geschwindigkeit von 600 km/h bis zu 25000 km ohne zusätzliche Betankung weit fliegen. Die tatsächliche Nutzlast liegt hierbei knapp unter einer Tonne.

Abbildung 2.3.: Global Hawk bei der Landung

Moderne UAV-Systeme (UAV: Unmanned Aircraft Vehicle) wie die Global Hawk sind weitgehend autonome Flugzeuge. Sie fliegen selbstständig, d.h. Lagekontrolle, Navigation und Systemkontrolle werden von den Bordcomputern durchgeführt. Der Flugzeugbediener verbleibt am Boden und hat im Normalbetrieb lediglich die „Richtlinienkompetenz“. Er autorisiert die Maschine zur Ausführung von Flugmanövern, definiert die Flugrichtung anhand von Wegepunkten und steuert die Sensoren. Mittels kombinierter Satelliten- und Trägheitsnavigation, mit der die Flugzeugposition auf weniger als einen Meter genau ermittelt wird, kann die Maschine unter jeglichen Sichtverhältnissen auf jedem Flugplatz eigenständig landen, der in der Datenbank gespeichert ist.

Momentan wird in der laufenden Produktion des Global Hawks an einer Leistungssteigerung und besseren Ausstattung, wie z.B. integrierten Selbstverteidigungssystemen gearbeitet. Um eine unökonomische Doppelentwicklung zu vermeiden, bringt die EADS (European

Aeronautic Defence and Space Company) im Austausch für den Global Hawk, das SIS (Signal Intelligence Sensorpaket) mit ein. In kurzer Zeit wird so der „Euro Hawk“ entstehen und auch in die Bundeswehr integriert werden.

2.4. MARVIN

Der ’Multi-prupose Aereal Robot Vehicle with Intelligent Navigation’ (MARVIN) ist ein autonomer Helikopter mit einem normalen Benzinmotor und einem Rotordurchmesser von 1,88 m. Das max. Startgewicht beträgt 11 kg.

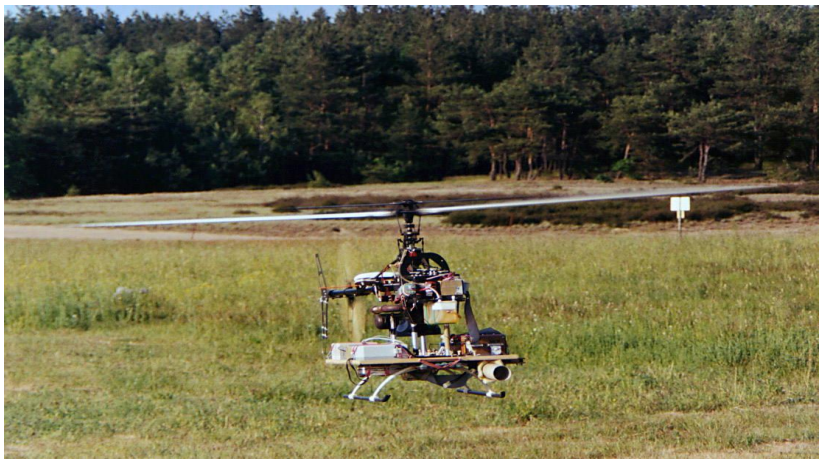


Abbildung 2.4.: MARVIN beim Testflug

Die Positionsbestimmung wird über GPS gehandelt und die Winkelbestimmung über einen selbstgebauten Beschleunigungssensor und einem gyroskopischen System. Ein Kompass zeigt die Position an, in der der Helikopter fliegt.

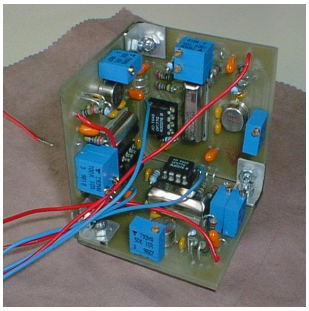


Abbildung 2.5.: Gyroskopisches System

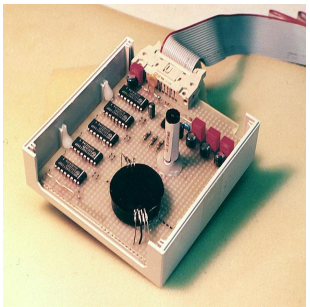


Abbildung 2.6.: Digitaler Kompass

Die Bilder werden durch eine nach unten gerichtete Kamera geschossen, über die serielle Schnittstelle an den Mikrocontroller weitergeleitet und dann an die Computer am Boden zur Auswertung geschickt. Dadurch entstehen bessere Bilder als beim Benutzen einer 2,4 Ghz Funkkamera. Beim Helikopter ist die Störung durch Verwackeln groß.

2.5. Microdrones

Wie den aktuellen Nachrichten im Mai zu entnehmen war [The Register: http://www.theregister.co.uk/2007/05/21/black_helicopters_over_merseyside/], verwendet die Stadt Liverpool in einer 3-monatigen Testphase fliegende Drohnen. Diese sollen laut Aussagen der Stadt zunächst nur für sicherheitsbezogene Überwachung verwendet werden. So z.B. die Überwachung von Parks und Demonstrationen. Später wäre eine Anwendung bei Sondereinsatzkommandos denkbar.

Abbildung 2.7.: Microdrones

Technische Details der Microdrones:

- vier Rotoren aus Carbon
- eingebauter Flightcontroller
- Gewicht ca. 900 g
- Durchmesser 70 cm
- Nutzlast 200 g

- zwei Akkus mit 14.8 V und 2300 mAh
- Flugdauer von ca. 20 min
- Reichweite für die Funksteuerung von 500 m
- Serielle Schnittstelle für externe Peripherie

Der eingebaute Flightcontroller sorgt dafür, dass der Pilot sich nicht um das Halten der Position/Lage kümmern muß. Der Flightcontroller stellt mit Hilfe eines „Ogpsmag“, einem GPS Ortungssystem, die Position fest. Gleichzeitig befindet sich im „Ogpsmag“ ein 3D-Magnetometer, worüber der Flightcontroller die Neigung und Schräglage bestimmen kann.

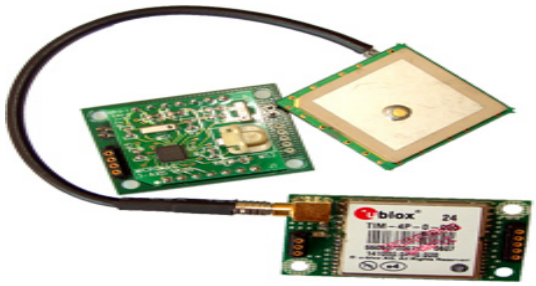


Abbildung 2.8.: Ogpsmag

Die Microdrones werden absolut professionell aufgebaut und können auch von ungeschulten Personen geflogen werden. Es ist wahrscheinlich, dass die Microdrones auf lange Sicht Ihre Anwendung im militärischen Bereich finden werden. Der deutsche Hersteller von Microdrones http://www.microdrones.com/ hat ein großes Sortiment an Peripheriegeräten, so dass sich die Anwendungsbereiche beliebig verfeinern lassen.

2.6. Lotte

An der Universität Stuttgart gibt es ein Projekt namens „Lotte“, das sich mit einem ferngesteuerten Outdoor Zeppelin beschäftigt. Die Dimensionen sind hier entsprechend groß und das Luftschiff muss von speziell ausgebildeteten Piloten ferngesteuert werden.

Alle wichtigen Informationen über den Flug wie Höhe, Geschwindigkeit und andere Statusinformationen werden über Funk an eine Bodenkontrollstation übermittelt.

Länge: 16 m

Gesamtvolumen: 109 m

Abbildung 2.9.: „Lotte“ im ferngesteuerten Flug

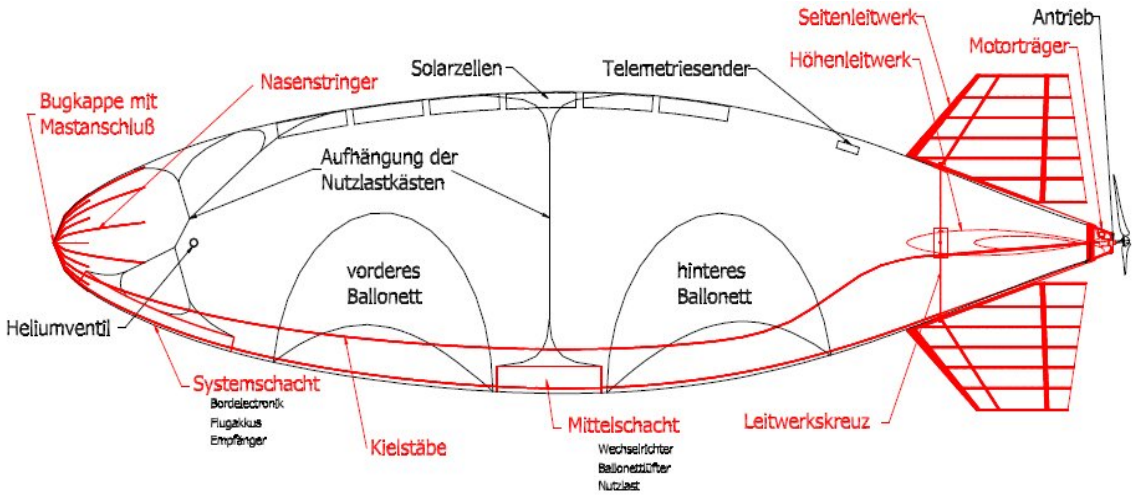


Abbildung 2.10.: Aufbau des Luftschiffs „Lotte“

Gewicht: 98 kg

Größter Durchmesser: 4 m

Max. Geschwindigkeit: 45 km/h

Max. Flughöhe: 1000 m

Nutzlast: 10-15 kg

Motorleistung: 1.500 W

Das Projekt „Lotte“ ist ein reines, nicht kommerzielles Forschungsprojekt. Es wurde unter Leitung eines Professors an der Uni Stuttgart konzipiert und von zahlreichen Studenten unterstützt und konstruiert. Aufgrund seines schadstoffarmen Antriebes wurde „Lotte“ schon häufig für Schadstoffmessungen eingesetzt, z.B. beim „Evaluierungsexperiment Augsburg “ (EVA), einem vom Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie (BMBF) finanzierten Forschungsprojekt.

Weitere interessante Auftritte hatte „Lotte“ auf diversen Openair Events (z.B. bei der „Internationalen Gartenbauausstellung“ (IGA) in Stuttgart), wo das Luftschiff als Werbeträger große Aufmerksamtkeit erregte.

2.7. ORCA - Observing Remote Controlled Airship

Das ORCA-Projekt, ebenfalls von der Universität Stuttgart konstruiert, beschreibt die Steuerung und Wegfindung eines teilautonomen Zeppelins im Indoor-Bereich. Der Zeppelin hat hierbei keine eigene Sensorik, sondern wird über ein aus mehreren Webcams bestehendes Kursverfolgungssystem koordiniert.

Der hierfür verwendete Zeppelin weist folgende Merkmale auf:

Ausmaße: 2,5 m Länge

Hülle: Polyethylen (PE)

Gas: Helium

Struktur: Carbonstangen

Nutzlast: 800 g

Antrieb: 4 Motoren

Steuerung: Kameraleitsystem

Dieses Projekt wurde im Rahmen einer Diplomarbeit realisiert, doch leider sind die meisten Quellen nicht mehr vorhanden. Das Projekt endete 2003.

Abbildung 2.11.: ORCA im Foyer der Uni-Stuttgart

Der Zeppelin sollte anhand eines Kameraleitsystems einen vordefinierten Pfad abfliegen können. Da das Flugschiff nicht mehr als 800 g Nutzlast mitnehmen kann, wurde auf eine Bordlogik verzichtet, und lediglich ein RC-Modul mit dazugehörigen Motoren für den Antrieb eingebaut.

Den groben Aufbau des Systems kann man folgender Graphik entnehmen:

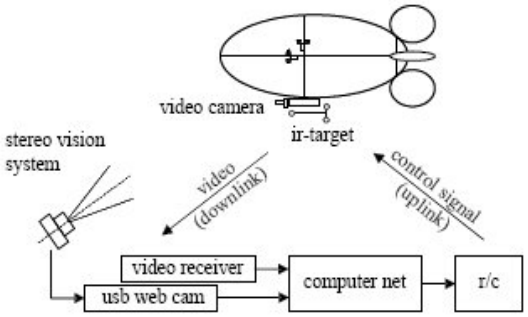


Abbildung 2.12.: ORCA: System-Aufbau

Eine vollständige autonome Steuerung wurde in einer zweiten Phase geplant, doch leider wurde dies nicht mehr umgesetzt. Somit kann davon ausgegangen werden, dass es kein öffentliches, gleichwertiges Projekt zu dem hier beschriebenen autonomen Zeppelin gibt.

Teil II.

Analyse

3. Autonome Flugobjekte in verschiedenen Umgebungen

Man muss grundsätzlich unterscheiden, ob ein autonomes Flugobjekt seine Umgebung kennt oder ob es in seinem Startzustand über keinerlei Wissen bezüglich seiner Umgebung verfügt.

Man spricht von einer bekannten Umgebung, wenn im Startzustand bereits Kenntnisse über die Umgebung, Hindernisse oder Zielobjekte vorhanden sind. Ein Roboter kann sich auf eine bekannte Umgebung schneller und besser einstellen, reagiert jedoch relativ unflexibel auf Änderungen seiner Umgebung. Hierbei kann es zu Problemen des Ablaufes durch äußere Einflüsse kommen. Man stelle sich ein Szenario vor, bei dem ein Roboter in einer Industriehalle eine Transportstrecke abfahren soll, wobei die Strecke vorher fest einprogrammiert wurde. Dabei besitzt der Roboter keine eigene Sensorik und kann somit seine Umgebung nicht weiter wahrnehmen. Wenn dieser Roboter nun zum Löschen einer Schiffsladung eingesetzt wird, kann dies verheerende Folgen haben.

Hat ein Roboter im Startzustand keinerlei Kenntnis über seine Umgebung, so besteht seine wichtigste Aufgabe darin, während der Ausführung seiner eigentlichen Aufgabe seine Umgebung wahrzunehmen und Hindernisse zu erkennen. Wenn er z.B. nicht a priori weiß, dass sich drei Meter hinter ihm eine Wand befindet, muss er diese mit Hilfe der Sensoren rechtzeitig erkennen, bevor er mit Selbiger kollidiert. Diese Erkennung muss höher priorisiert sein als die Durchführung seiner eigentlichen Aufgabe.

Die Umgebung eines Agenten lässt sich im Wesentlichen durch die Dimension des Raumes und seine Beschaffenheit klassifizieren.

In einer effizienten Struktur ist die Beschaffenheit der Umgebung für den Roboter von zweitrangiger Bedeutung. Die Umgebung ist nur für die Art des Antriebs wichtig, nicht aber für das abstrakte Level der Programmierung. Wenn in einem Programmteil eine spezielle Anweisung zur Erhöhung der Geschwindigkeit vorkommt, wird damit nur ein abstrahiertes Steuerinterface angesprochen nicht aber die maschinennahen Steuerelemente selbst. Wichtiger sind hierbei die äußeren Einflüsse, die auf ihn einwirken. Im Wasser wäre das z.B. die Strömung oder in der Luft Windböen. Diese müssen in jeder Umgebung unabhängig von der Anzahl der Dimensionen individuell Berücksichtigung finden.

3.1. Zweidimensionaler Raum

Im zweidimensionalen Raum bewegt sich ein Roboter nur auf einer Ebene (X - und der Y-Achse). Er kann maximal mit einem anderen Hindernis kollidieren oder anhalten, wenn er eine Fehlfunktion oder keine Versorgungsspannung mehr hat.

Als Beispiel kann man hier direkt auf den Hamburger Hafen verweisen. Auf dem neuen Container Terminal in Hamburg-Altenwerder werden die Schiffe mit autonomen Container-Brücken (Kränen) entladen. Diese sind fest auf Schienen verankert. Die Ladung wird angehoben, zur gewünschten Stelle transportiert und dort herabgelassen. Hier kann man also von einem Agenten sprechen, dessen Fortbewegung in zweidimensionaler Umgebung stattfindet.

3.2. Dreidimensionaler Raum

Im dreidimensionalen Raum stellt sich das Ganze komplexer dar. Der Agent muss eine zusätzliche Dimension berücksichtigen - sowohl in Bezug auf seine zu erfüllende Aufgabe als auch in Bezug auf Fortbewegung und Hindernisse.

Als Beispiel sei hier auf den Zeppelin verwiesen, den wir im Rahmen unserer Bachelorarbeit gebaut haben. Dieser hat den Vorteil, dass er durch das Helium eigenständig die Höhe beibehält und nur gelegentlich korrigieren muss. Gäbe es einen Spannungsverlust, würde der Zeppelin nicht sofort abstürzen und beschädigt werden, sondern er würde zunächst bewegungsunfähig schweben und kann ohne Probleme geborgen werden. Dies wäre bei einem autonomen Hubschrauber oder Flugzeug nicht der Fall. Käme es hier zu einem Problem mit der Stromversorgung, würde der Helikopter umgehend abstürzen. Das Flugzeug kann in diesem Fall höchstens noch einen kontrollierten Gleitflug einleiten.

> Dies hat zur Folge, dass bei einem Agenten im dreidimensionalen Raum Systeme, zur Erhaltung der Höhe und Erkennung von Hindernissen oberste Priorität bekommen müssen.

Folglich wirkt sich die Umgebung unmittelbar auf die Beschaffenheit und Komplexität des Agenten aus. Abgesehen von der unterschiedlichen Struktur und dem äußeren Aufbau, benötigt ein Agent im Wesentlichen mehr Hardware im dreidimensionalen Raum als im zweidimensionalen Raum. Im dreidimensionalen Raum wird eine erhöhte Anzahl an Sensoren zur Erfassung der zusätzlichen Umgebungsdaten und ein zusätzlicher Motor zur Erhaltung der Höhe benötigt. Bei der Software verhält es sich ähnlich. Eine Programmroutine zur Bestimmung des Neigungswinkels, wie es für einen autonomen Hubschrauber nötig wäre, ist

komplexer als eine einfache Abfrage der Entfernung zu einem in zwei Metern Entfernung befindlichen Hindernis in derselben Ebene.

3.3. Indoor-Flugobjekte im Vergleich zu Outdoor-Flugobjekte

Im Verlauf der Konstruktion des Zeppelins musste eine Entscheidung für Indoor oder Outdoor als Zielumgebung getroffen werden.

Der wesentliche Unterschied hierbei besteht nicht in der Umgebung an sich, sondern in den Einflüssen, die auf den Roboter einwirken.

	Indoor	**Outdoor**
Luftböen	Durch sich plötzlich öffnende Türen oder Tore können unerwartete Luftströmungen entstehen.	Unvorhersehbare Windböen können das Flugobjekt unkontrolliert herumwirbeln.
Temperatur und Luftfeuchtigkeit	Größtenteils konstant und vorhersagbar; das Flugobjekt kann darauf eingestellt werden.	Nur bedingt konstant und nicht vorhersagbar; Regenschauer oder starke Hitze können Material überbeanspruchen.
Räumliche Begrenzung	Durch räumliche Begrenzung bleibt das Flugobjekt immer in Reichweite.	Das Flugobjekt könnte bei fehlerhaften Einstellungen weit wegfliegen und dadurch unkontrollierbar werden.

Tabelle 3.1.: Indoor vs. Outdoor

Nach eingehender Gegenüberstellung der Indoor- und Outdooreigenschaften sind wir im Rahmen dieser Arbeit zu dem Entschluss gekommen, den Zeppelin für den reinen Indoorbereich zu konzipieren. Zum einen, weil seine Hauptaufgabe darin bestehen sollte, in Messehallen „werbewirksam aufzufallen“, und zum anderen, da eine aerodynamische Form in unserem Fall nicht möglich ist. Das Volumen muss sehr groß sein, um ausreichenden Auftrieb sicher zu stellen, was wiederum zu einer hohen Kantenlänge bzw. Seitenfläche führt. An dieser Stelle sei auf unsere Vorausberechnungen in Kapitel 5 verwiesen. Ein Zeppelin in diesen Dimensionen würde über eine sehr große Angriffsfläche gegenüber dem Wind verfügen und müsste daher im Outdoor-Bereich über einen sehr starken eigenen Antrieb verfügen. Ohne diesen Antrieb wäre eine voll autonome Steuerung nicht möglich, da man ständig extern eingreifen müsste, um die Sicherheit des Systems zu gewährleisten. Je leistungsfähiger ein Motor ist, desto größer und schwerer wird er. Ein schwerer Motor erhöht

das Eigengewicht des Flugobjektes und muss durch ein noch größeres Gasvolumen ausgeglichen werden. Dies führt wieder zu eine vergrößerten Oberfläche. Man kann also von einer Art „Teufelskreis“ sprechen.

Im universitären Umfeld sind gewisse Einschränkungen und Voraussetzungen zu erfüllen, die im nachfolgenden Kapitel erörtert werden.

4. Anforderungen von autonomen Flugobjekten im universitären Umfeld

4.1. Generelle Einschränkungen

Autonome Flugobjekte im universitären Umfeld sind ähnlich aufgebaut wie diejenigen aus Militär, Industrie und Forschung, wobei zusätzliche Einschränkungen berücksichtigt werden müssen. Diese Einschränkungen gelten insbesondere für eine Bachelorarbeit. Diese Einschränkungen betreffen:

- Finanzielle Mittel
- Materialien
- Zeitlicher Rahmen
- Räumlichkeiten / Campus
- Wiederverwendbarkeit

4.1.1. Finanzielle Mittel

Die Berücksichtigung des eingeschränkten Budgets stellt bei der Konzeption und dem Aufbau eines autonomen Flugobjektes die größte Herausforderung dar. Gewisse Materialien, die zum Bau benötigt werden, sind kostspielig, da es sich hierbei um kleine, leichte Komponenten handelt, die das Gewicht auf ein Minimum reduzieren sollen. Es gibt beispielsweise sehr kleine Servomotoren, die bei gleicher Kraft gegenüber größeren Modellen das Doppelte kosten. Gleiches gilt für sämtliche andere Komponenten wie z.B. Akkus, Mikrocontroller, Motoren oder Lüfter. Um den finanziellen Rahmen einer Abschlussarbeit nicht zu sprengen, sollte man daher einen Mittelweg finden, zwischen dem qualitativ Besten und dem, was das Budget hergibt.

Bei der Entwicklung eines Zeppelins ist das Gewicht sogar noch ausschlaggebender als bei anderen Flugobjekten. Je mehr Tragkraft der Zeppelin am Ende haben soll, desto mehr

Helium benötigt man, und desto größer wird wiederum der Zeppelin. Je größer der Zeppelin, desto mehr Eigengewicht hat er, da die Gesamtkonstruktion mitwächst. Nur durch sehr kleine Komponenten mit geringem Eigengewicht kann man einen relativ kleinen Zeppelin bauen. In unserem Fall ist die Größe des Zeppelins in sofern beschränkt, da der Zeppelin nur innerhalb von Gebäuden fliegen soll. Dies führt unweigerlich dazu, dass die Materialien eines Zeppelins sehr teuer sind.

4.1.2. Materialien

Die Konstruktionsmaterialien, die für ein autonomes Flugobjekt in Frage kommen, sind sehr speziell. Auch hierbei spielt das Gewicht eine wichtige Rolle. Soll eine Holzkonstruktion verwenden werden, kommt nur Balsaholz wegen seiner geringen Dichte in Frage. Bei Metallen sollte man darauf achten, dass man ebenfalls Aluminium verwendet, aufgrund seiner geringen Dichte. Leider lässt sich Aluminium nur sehr schwer verarbeiten, was einen zusätzlichen Zeitaufwand bedeutet. Wird auch noch Wert auf ein gewisses Design gelegt, so erschwert sich die Entwicklung umso mehr. Einige Materialien stammen aus der Raumfahrt und sind nicht nur extrem teuer, sondern zudem schwer erhältlich. Allerdings haben solche Materialien meistens die besten Eigenschaften für die Luftfahrt.

4.1.3. Zeitlicher Rahmen

Da eine Abschlussarbeit zwischen 3 und 6 Monaten dauert, ist man bei der Entwicklung, beim Aufbau, Testen und Schreiben der Abschlussarbeit eingeschränkt. Durch die kurze Zeit ist man gezwungen einen genauen Plan zu erstellen und jede Phase genau einzuhalten. Man sollte sich nicht mit zu vielen Themen gleichzeitig beschäftigen, da die Zeit nicht ausreichen wird. Es ist immer besser ein Gerät mit wenigen Fähigkeiten zu bauen, welches am Ende aber sein Ziel erreicht, als ein ausgefallenes Gerät, welches nur ansatzweise fertig geworden ist.

4.1.4. Räumlichkeiten / Campus

Ein weiterer wichtiger Punkt, der beim Bau eines autonomen Flugobjektes berücksichtigt werden muss, sind die Räumlichkeiten in und an einer Hochschule. Aufgrund des hohen Platzbedarfs kann man also sagen, dass die meisten autonomen Flugobjekte für den Außenbereich konzipiert werden. Einen Helikopter kann man schlecht im Flur oder in einem Zimmer betreiben. Ein Flugzeug braucht Platz zum Starten und Landen. Ein Zeppelin benötigt am meisten Platz, da der Auftrieb, der hier durch Helium erzielt wird, stimmen muss. Bei

3 kg Gesamtgewicht, verbraucht man ca. 3 m³ Helium. Das bedeutet, dass man 3 m³ allein für den Zeppelin braucht, ohne seine Flugroute mit einzuberechnen.

4.1.5. Wiederverwendbarkeit

Unter Wiederverwendbarkeit verstehen wir im Rahmen einer Abschlussarbeit, die Möglichkeit, für andere Studierende im Anschluss das Flugobjekt erneut zu gebrauchen, um vielleicht eine darauf aufbauende Abschlussarbeit zu konzipieren. Die Einschränkung hierbei stellen die benutzten Materialien dar, die auch über einen längeren Zeitraum standhalten müssen. Der Aufbau muss so erfolgen, dass auch neue Features eingebaut werden können. Also sollte von vorneherein z.B. mehr Tragkraft einberechnet werden.

Desweiteren sollte man nicht versuchen, alles was technisch möglich ist, umzusetzen, sondern nachfolgenden Abschlussarbeiten die Möglichkeit bieten, auf dieser Arbeit aufzubauen.

4.2. Anforderungen an die Hardware

4.2.1. Komponenten

Die Hardware eines Flugobjektes besteht aus Konstruktionsmaterialien sowie aus elektrischen Komponenten.

Die Konstruktionsmaterialien variieren je nach Flugobjekt. Eine Eigenschaft müssen aber alle Materialien erfüllen unabhängig vom Typ. Sie müssen ein geringes Gewicht bzw. eine geringe Dichte aufweisen. Dabei müssen die Materialien allerdings widerstandsfähig sein, damit sie nach einem möglichen Absturz nicht sofort unbrauchbar werden. Am besten baut man das Objekt modular auf, so dass man es einfach zum Transport auseinander und wieder zusammenbauen kann (insbesondere bei sperrigen Objekten).

Die benötigten elektrischen Komponenten kann man wiederum in vier Kategorien einteilen:

- Sensoren
- Aktoren
- Controller und
- Energiequelle.

4.2.2. Sensoren

Sensoren sind die Komponenten, die dem Flugroboter als Eingangsgrößen dienen. Über die Sensoren hat der Roboter Zugang zu seiner Umwelt. Die Eingangsgrößen werden nach Erhalt von dem Controller (Logikeinheit) verarbeitet und anschließend an Aktoren in Form von Befehlen weitergegeben. Beispiele für Sensoren:

- Sharp-Infrarot-Sensoren
- Ultraschall-Sensoren
- Bluetooth-Modul (sowohl Sensor als auch Aktor)
- Bumper (z.B. aus Gitarrenseiten)
- Video/Fotokamera

4.2.3. Aktoren

Aktoren sind die ausführenden Komponenten. Durch Aktoren kann der Roboter Einfluss auf seine Umwelt nehmen. Je nachdem welche Befehle die Aktoren von der Logikeinheit bekommen, werden unterschiedliche Aktionen ausgeführt. Beispiele für Aktoren:

- Motoren
- Servo-Motoren
- Lüfter
- Lautsprecher
- Bluetooth-Modul (sowohl Sensor als auch Aktor)

4.2.4. Controller

Controller oder auch Mikrocontroller übernehmen die Entscheidungen in einem Sense-Act-Prozess. Erst durch die programmierte Logik ist es möglich auf die Umwelt kontrolliert zu reagieren.

Mikrocontroller besitzen meistens mehrere analoge/digitale Eingänge und Ausgänge, sowie eine Kommunikationsschnittstelle, über die man sie programmieren kann und über die man auch während des normalen Betriebes Informationen austauschen kann.

Es wurde in dieser Arbeit einen speziell entwickelten Mikrocontroller der Partner-Fachhochschule Brandenburg benutzt, das so genannte Aksenboard. Dieses Board besitzt sämtliche Schnittstellen, die wir für unseren Flugroboter benötigen und sogar noch mehr. Wir haben uns für die Arbeit mit diesem Board entschieden, da uns das Board kostenlos zur Verfügung stand, die Programmierschnittstelle sehr gut dokumentiert ist und sämtliche Funktionen implementiert werden können. Punkte, die gegen dieses Board sprachen, waren das erhöhte Gewicht und die Außenmaße (zum Teil wegen der extra Funktionen, die wir eigentlich nicht benötigten).

4.2.5. Energiequelle

Als Energiequelle für einen Flugroboter sollte man auf die neusten Entwicklungen Rücksicht nehmen. Zu empfehlen sind hier Li-Polymer-Akkus und das aus mehreren Gründen:

- Sehr Platzsparend
- Konstanter, ergiebiger Gleichstrom
- Wenig Gewicht
- (schnell) Wiederaufladbar

Leider sind die Kosten dafür sehr hoch. Man benötigt aber nicht nur die Akkus, sondern auch ein spezielles Ladegerät. Die Li-Polymer-Akkus können auf kleinstem Raum große Mengen Energie freigeben. Da man das Flugobjekt auch testen und betreiben will, ist es nötig (nicht nur aus ökologischen Gründen) wiederaufladbare Akkus einzusetzen. Da es bei Flugobjekten sehr auf das Gewicht ankommt, benötigt man eine Energiequelle, die die oben beschriebenen Eigenschaften erfüllt. Schließlich soll sich das Gerät entsprechend lange in der Flugphase befinden.

Eine externe Energiequelle, die die Stromversorgung über ein Kabel realisiert, ist zwar denkbar, doch bei Flugobjekten ungünstig und nicht empfehlenswert, da durch ein Kabel die Flugreichweite sehr eingeschränkt wäre.

4.2.6. Aerodynamik

Aerodynamik sollte an dieser Stelle vollständigkeitshalber erwähnt, aber nicht weiter ausbaut werden. Dies hat den einfachen Grund, dass das Flugobjekt „Zebus“ keine Ansprüche an ein besonderes aerodynamisches Design erfüllen muss und dieses Thema den Rahmen dieser Arbeit sprengen würde.

Bei Helikoptern und Flugzeugen ist dieses Thema von größerer Bedeutung. Indoor-Zeppeline besitzen zwar ebenfalls gewisse Designvorgaben, doch sind sie gegenüber Luftströmen wesentlich unempfindlicher. Das liegt daran, dass der Auftrieb des Zeppelins nicht durch Aerodynamik erbracht wird, wie dies bei Helikoptern und Flugzeugen der Fall ist, sondern durch Helium (bzw. Wasserstoff). Darüber hinaus bewegt sich ein Zeppelin wesentlich langsamer im dreidimensionalen Raum als andere Flugobjekttypen, weshalb man die Aerodynamik im Rahmen dieser Arbeit gänzlich vernachlässigen kann.

4.3. Anforderungen an die Software

4.3.1. Allgemeines

Ein autonomes Flugobjekt wie beispielsweise ein autonomer Hubschrauber, stellt an seine Steuerungssoftware wesentlich höhere Anforderungen als ein zweidimensionaler Mindstorm Roboter.

Die entscheidende Rolle hierbei spielt die Umgebung. Je extremer die Umgebung, desto höher werden die Ansprüche an die Software. Ein Outdoor- Flugobjekt, dass bei jeglichen Witterungsverhältnissen fliegen soll, muss in seinen Programmroutinen sehr viel mehr berechnen und mehr Sensorwerte verarbeiten als ein Roboter der Mindstorm Serie.

Ein Roboter wie beim „Robocup Wettbewerb“ muss sich während seines Lebenszyklus lediglich auf maximal zwei Dimensionen konzentrieren.

Ein autonomer Hubschrauber muss beispielsweise auf alle drei Dimensionen gleichzeitig achten. Ein Fehler bei der Höhen- und/oder Lagerkontrolle könnte den Absturz bedeuten. Hinzu kommt, dass ein autonomes Flugobjekt wesentlich größere Datenmengen in kürzerer Zeit verarbeiten muss.

Daraus könnten sich folgende Anforderungen für die Steuerungssoftware eines autonomen Flugobjekts im Indoorbereich ergeben wie nachfolgend am Beispiel eines Zeppelins beschrieben.

4.3.2. Prioritätenvergabe für einzelne Lebenserhaltungstasks

- Prio 5: Überwachung des Auftriebs
- Prio 4: Abstand zum Boden
- Prio 3: Abstand zur Decke

- Prio 2: Abstand zu einem vorderen/hinteren Hindernis
- Prio 1: Abstand zu einem seitlichen Hindernis
- Prio 0: Missionsziele

Die Tasks sind nach Relevanz geordnet, wobei diejenigen die oberste Priorität erhalten, die für den reibungslosen Flug am wichtigsten sind. Die Task, die den Auftrieb überwachen soll, muss beispielsweise höchste Priorität genießen. Es muss einer ungewollten Veränderung des Auftriebs umgehend entgegengewirkt werden, sonst droht der Absturz.

4.3.3. Subsumption Modell

Einfache Aktionen können durch komplexere Aktionen überlagert werden. Zunächst werden die Basisfunktionen implementiert, denen später komplexere Aktionen hinzugefügt werden können. Durch Subsumption ist eine klare und leicht erweiterbare Struktur gegeben.

4.3.4. Überlebenswichtige Funktionen

Die überlebenswichtigen Funktionen sollten atomar und nicht preemptiv sein. Wenn eine Task aus einem höheren Level (z.B. Kamerafunktionen zur Bildmustererkennung) eine Task aus einem unteren Level (z.B. Kontrolle des Abstandes zum Boden) in Ihrer Rechenzeit unterbrechen darf, führt dies unweigerlich zu Störungen. Diese äußern sich z.B. im Sinkflug.

Der Zeppelin muss laufend die Höhe kontrollieren, wobei eine Unterbrechung durch die Kameraroutinen hervorgerufen wird. Der Höhenkontrolle steht dadurch weniger Rechenzeit zur Verfügung. Das Flugverhalten wird nicht mehr vorhersagbar und eine Kollision möglich.

4.3.5. Verarbeiten der Daten

Eine schnelle, effiziente und direkte Datenverarbeitung wird durch einen kurzen Weg zwischen Sensoren und Aktoren sowie schnelle Zwischenspeicherung erreicht.

Da Platz und Kapazität bei einem autonomen Flugobjekt begrenzt sind, muss die Hardware dementsprechend angepasst bzw. ausgewählt werden. Die Software sollte ebenfalls effizient programmiert werden. Es gilt überflüssige Schleifen wie auch falsch dimensionierte Datenstrukturen zu vermeiden, um die Anforderungen an die Hardware und damit auch den benötigten Platzbedarf zu minimieren.

4.3.6. Reaktionszeiten

Ein RTOS muss minimale Reaktionszeit für Aktionen (zeitkritische Prozesse) garantieren. Die Wichtigkeit der lebenserhaltenden Tasks wurde bereits erwähnt. Bei Ihnen ist eine minimale Reaktionszeit unabdingbar. Müsste die Steuerungssoftware erst mehrere Sekunden auf das Ergebnis der Höhenmessung warten, so wäre es nach Berechnung der entsprechenden Reaktion vielleicht schon zu spät, um zu reagieren.

4.3.7. Schnittstellen

Klare Schnittstellen zwischen Steuerungssoftware und Hardware, ähnlich wie im Praktikum „Prozesslenkung“, werden über mehrere Ebenen abstrahiert. Auf der unterster Ebene sollte der Treiber stehen, der hardware-spezifisch Sensoren und Aktoren anspricht. Dieser Treiber könnte von der grafischen Oberfläche aus über festgelegte Schnittstellen angesprochen werden.

4.3.8. Kernel

Auf dem Controllerboard sollte sich möglichst ein MicroKernel mit einem RTOS befinden. Ein Microkernel hat den Vorteil, dass in ihm nur die elementarsten Module vorhanden sind. Diese können dann ganz speziell auf die Bedürfnisse des Host Systems angepasst werden.

4.3.9. Visualisierungssoftware

Die Visualisierungssoftware sorgt dafür, dass der Zeppelin seine Umgebung erfassen kann. Abgesehen von den Sensoren, die die Aufgabe haben, den Zeppelin am Leben zu halten, soll die Visualisierungssoftware die Durchführung der eigentlichen Aufgabe sicherstellen.

Visuelle Missionsziele müssen erkannt und über festgelegte Schnittstellen an die Steuerungssoftware gesendet werden. In dieser Arbeit, die im universitären Umfeld erbracht wurde, war der Kostenpunkt von entscheidender Bedeutung. Daher lag die Entscheidung für das OpenSource Paket „ARToolkit“ nahe. Es erfüllt die gegebenen Anforderungen, ist frei verfügbar und durch die Java Schnittstelle „JARToolkit“ plattformunabhängig. Die Kommunikation zwischen Visualisierungssoftware und Steuerungssoftware ist nur über eine bekannte Schnittstelle möglich. In diesem Rahmen dient die Programmiersprache Java als kommunikationsfördernd, da sie universell einsetzbar ist und die bereits erwähnten positiven Eigenschaften mit sich bringt.

5. Grundidee und Konzeption eines autonomen Zeppelin

Die Idee zu einem „autonomen Zeppelin“ kam uns während des Hochschulprojekts „Mobile Roboter“. Wenn es möglich ist, einen „Legoroboter“ mit einer Hinderniskontrolle auf dem Boden fahren zu lassen, warum entwirft man dann nicht ein autonom fliegendes Flugobjekt, das lediglich eine Dimension mehr, die Höhe, berücksichtigen muss? Der naheliegendste Gedanke war, einen Helikopter zu benutzen, da dieser auf der Stelle fliegen kann und im Gegensatz zu einem Flugzeug nur einen minimalen Raum zum Fliegen benötigt (dazu kommen beim Flugzeug noch die komplexen Feinheiten wie Aerodynamik und Antrieb). Erste Recherchearbeiten ergaben einige Aufzeichnungen zu Studien und Versuchen mit „autonomen Hubschraubern“. Bei allen Projekten bestand die Hauptschwierigkeit im Wesentlichen darin, die Ausrichtung des Helikopters beizubehalten. Diese Problemstellung und der ziemlich hohe Energiebedarf für den Antrieb, führten zu dem Entschluss, für den Auftrieb Helium zu verwenden und so einen sparsameren Antrieb zu erhalten.

Zur Überprüfung der Realisierbarkeit der Ideen, wurden einige Vorberechnungen im Vorfeld durchgeführt.

Die Bachelorarbeit von Steffen Hinck und Andreas Piening bestätigte das Vorhaben. Deren Problematik bestand im weitesten Sinne darin, ein Modul zu bauen, was die Ausrichtung/Schräglage des Helikopters erkennt. Hinzu kommt die Erkenntnis, dass die Flugdauer eines solchen Helikopters nicht sehr ausdauernd ist.

5.1. Hardware Aufbau eines Zeppelins - Entwürfe, Komponentenbeschreibung

Die Entwicklung eines Zeppelins hängt in erster Linie von dem späteren Einsatzbereich des Gerätes ab. Darüber hinaus sind viele Faktoren zu beachten, die bereits in den generellen Anforderungen ansatzweise beschrieben wurden. Zu den wichtigsten Punkten bei der Konzeption eines autonomen Zeppelins zählen das Design, die Steuerung, die Wahrnehmung der Umwelt sowie die Künstliche Intelligenz.

5.1.1. Vorberechnungen

Aus materialtechnischer Gründe fiel die Entscheidung auf eine Kantenlänge von 1,30 m. Daraus resultiert ein Volumen für die Ballonhülle von 1,30 m x 1,30 m x 1,30 m = 2,197 m³.

Komponente	Gewicht [g]
Ballonhülle (10,14 m²)	304
Balsaholzkonstruktion	506
Fieberglaskabelschächte	100
Kabel, Schrauben etc.	80
Aksenboard	110
4 Li-Polymer-Akkus	102
Funkkamera	20
6 Ultraschallsensoren	120
2 Lüfter zur Steuerung inkl. Servomotor	140
Bluetooth Seriell Modul	67
Summe	**1549**

Tabelle 5.1.: Komponentengewichte

Der Auftrieb muss stark genug sein, um die 1549 g Nutzlast tragen zu können. Das verwendete Trägergas Helium hat einen Auftrieb von 1046 g/m³. Die konzipierte Ballonhülle mit einem Volumen von ca. 2,2 m³ könnte also eine Nutzlast von 2,3 kg transportieren.

Die Nutzlast von 1,55 kg kann daher mit der nach diesen Berechnungen konstruierten Ballonhülle ohne Probleme bewegt werden. Der resultierende „Gewichtspuffer“ von 2,3 kg - 1,55 kg = 0,75 kg erschien notwendig. Da die Ballonhülle eine komplette Eigenkonstruktion ist, lassen sich Bearbeitungsfehler im Hinblick auf die Gasdurchlässigkeit und der eventuell auftretenden Größenunterschiede ausschließen.

5.1.2. Design eines Zeppelins

Das Design eines Zeppelins hängt sehr stark von der späteren Verwendung ab. Das Ziel besteht darin, einen Zeppelin für den Indoorbereich zu konzipieren. Daher sollte die Aerodynamik vernachlässigt werden und die äußere Form des Zeppelins etwas freier gewählt werden können. Der geplante Einsatzort des Zeppelins soll sich innerhalb von Messehallen befinden, mit dem Zweck, deren Besucher werbetechnisch und medienwirksam zu beinflussen. Die Vermittlung von Werbebotschaften erfolgt durch die Nutzung der Oberfläche des Zeppelins. An dieser Werbefläche können jegliche Art von audiovisuellen Maßnahmen angebracht werden.

Aufgrund dieser Tatsache wurde die Figur eines Kubus gewählt. Diese Form ist zwar nicht sehr aerodynamisch, doch bieten die vier großen Seitenflächen genug Platz für Werbebanner.

Abbildung 5.1.: Zeppelin: Kubus

Die Konstruktion des Zeppelins kann man in zwei Bereiche unterteilen:

- eine geschlossene Hülle für den Auftrieb und
- einen Unterbau für die Steuerung und die elektrischen Komponenten

Die Hülle

Die Hülle des Zeppelins muss trotz der großen Fläche extrem leicht sein. Dabei sollte sie allerdings Reißfestigkeit und eine große Dichte besitzen, damit das Helium-Gas nicht zu schnell aus der Hülle entweichen kann. Daher wurde ein Material aus der Luft- und Raumfahrt ausgewählt - Mylarfolie.

Mylarfolie ist zwar im Gegensatz zu herkömmlichen Materialien teuer und schwieriger zu verarbeiten, doch besitzt sie eine hohe Reißfestigkeit und ein extrem geringes Eigengewicht (30 g/m²). Die Dichte des Materials ist gleich zu setzen mit PVC- oder PE-Folie, das Austreten von Helium aus der verschweißten Folie wird dadurch auf ein Minimum reduziert.

Abbildung 5.2.: Zeppelin: Hülle - Mylarfolie

Der Unterbau

Der Unterbau des Zeppelins soll später alle elektrischen Komponenten beinhalten, die für die Erfassung und Beeinflussung der Umwelt verantwortlich sind.

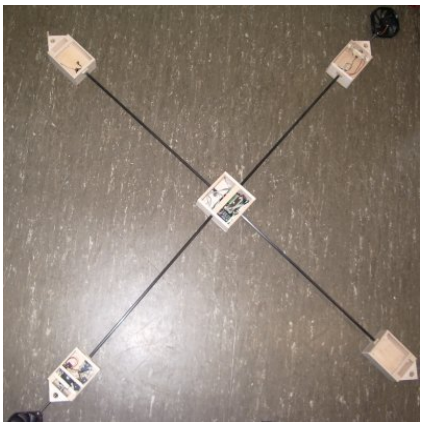


Abbildung 5.3.: Zeppelin: Unterbau

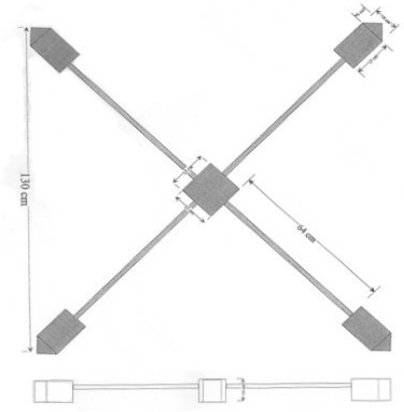


Abbildung 5.4.: Zeichnung des Unterbaus

Diese stellen das Hauptgewicht des gesamten Zeppelins dar und sollen daher beim Modell unterhalb der Hülle hängen. Dadurch befindet sich der Schwerpunkt unten. Im Falle eines Sturzes muss die Struktur des Zeppelins so konstruiert sein, dass die Komponenten geschützt werden.

Bei der Materialauswahl für den Unterbau fiel die Entscheidung aus den folgenden Gründen auf Balsaholz: Es ist sehr leicht zu verarbeiten und schützt die Komponenten optimal. Außerdem ist Balsaholz vom Gewicht her sehr leicht.

Um die Last optimal unter der Hülle zu verteilen, wurde die Konstruktion modulartig aufgebaut:

- 4 Eck-Module
- 1 Mittel-Modul

Eck-Module

Die Eck-Module werden, wie der Name bereits vermuten lässt, an den vier unteren Ecken der Hülle befestigt. Zwei von den Modulen beinhalten Servomotoren, an denen über eine Achse jeweils ein Lüfter verbunden ist. In dem dritten Modul befindet sich die Bluetoothkomponente und in dem vierten Modul die Energiequelle (Li-Polymer-Akkumulatoren).

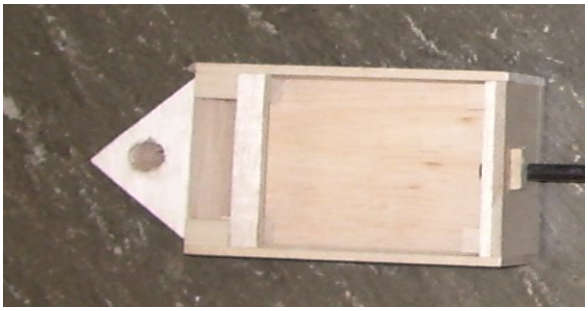


Abbildung 5.5.: Zeppelin: Eck-Modul

Mittel-Modul

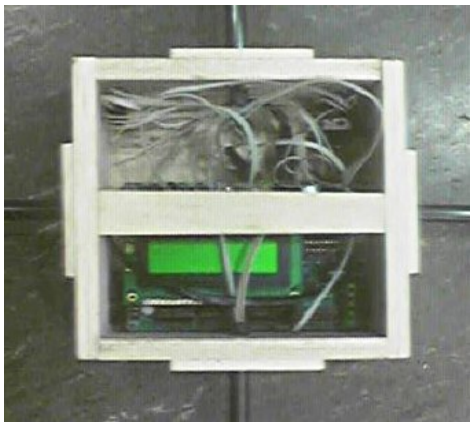


Abbildung 5.6.: Zeppelin: Mittel-Modul

Das Mittel-Modul wird an der Unterseite der Hülle mittig ausgerichtet. Es beinhaltet den Mikrocontroller, der den Zeppelin steuert und mit der Bodenstation (Laptop) kommuniziert.

Des Weiteren befindet sich im Mittel-Modul eine RF-Kamera, deren Bilder per Funk an die Bodenstation zur Auswertung weitergeleitet werden.

Die Module sind durch Carbonrohre miteinander verbunden, durch die später die Kabel geführt werden.

5.1.3. Steuerung eines Zeppelins

Die Steuerung des Zeppelins wird primär durch zwei Lüfter realisiert. Diese sind über eine Achse mit Servomotoren verbunden, die eine Drehung der Lüfter ermöglichen. Die Lüfter können in einem Winkel von 180° bewegt werden.

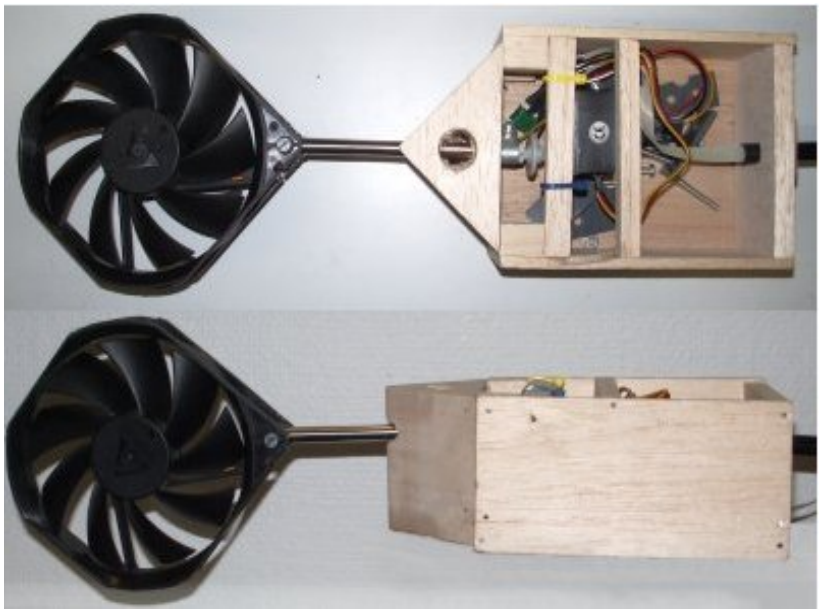


Abbildung 5.7.: Zeppelin: Eck-Modul mit Steuerung

Der Zeppelin kann also, je nachdem welchen Winkel die Servomotoren einnehmen, vorwärts- und rückwärts fahren, sich auch auf der eigenen Achse drehen sowie steigen. Der Zeppelin sollte auch sinken, doch ist es mit einem 180° Servomotor, nicht möglich. Andere Motoren standen nicht zur Verfügung. Außerdem ist davon auszugehen, dass der Zeppelin nach kurzer Zeit so viel Helium verliert, dass der Sinkflug automatisch erfolgt wenn keine Gegenmaßnahmen einleitet werden.

5.1.4. Wahrnehmung der Umwelt

Die Wahrnehmung der Umwelt des Zeppelins erfolgt über sechs Ultraschall-Sensoren. Diese werden in der Mitte von jeder Fläche der Kubushülle befestigt und messen die Distanz zu Objekten, die vom Zeppelin als Hindernisse erkannt werden. Die Sensoren sind über einen I²C-Bus mit dem Mikrocontroller verbunden, der im Falle des Auftretens einer Barriere reagiert und den Zeppelin in eine andere Flugbahn lenkt.

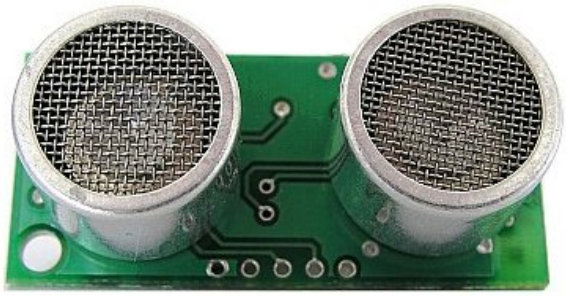


Abbildung 5.8.: Zeppelin: Ultraschall-Sensor

5.1.5. „Gehirn“ eines Zeppelins

Als Gehirn des Zeppelins wird die Einheit bezeichnet, die die Aktoren und Sensoren koordiniert einsetzt, um einen reibungslosen Flug des Zeppelins zu ermöglichen. Dabei werden die Entscheidungen von zwei verschiedenen Orten/Komponenten getroffen:

- Der Mikrocontroller an Bord des Zeppelins und
- die Bodenstation (Laptop)

Der Mikrocontroller

Der Mikrocontroller trifft Entscheidungen, die während des Fluges für Stabilität sorgen. An den Mikrocontroller sind sämtliche Aktoren und Sensoren des Zeppelins angeschlossen mit Ausnahme der RF-Kamera.

Die Bodenstation

Die Bodenstation wiederum steht per Bluetooth in Verbindung mit dem Mikrocontroller. Die Kamera operiert per Funk mit der Bodenstation. Die Videosignale werden von der Kamera direkt an die Bodenstation gesendet. Dort werden sie über eine Framegrabberkarte empfangen und verarbeitet. Wenn nötig, wird per Bluetooth ein neuer Flugbefehl an den Mikrocontroller weitergeleitet.

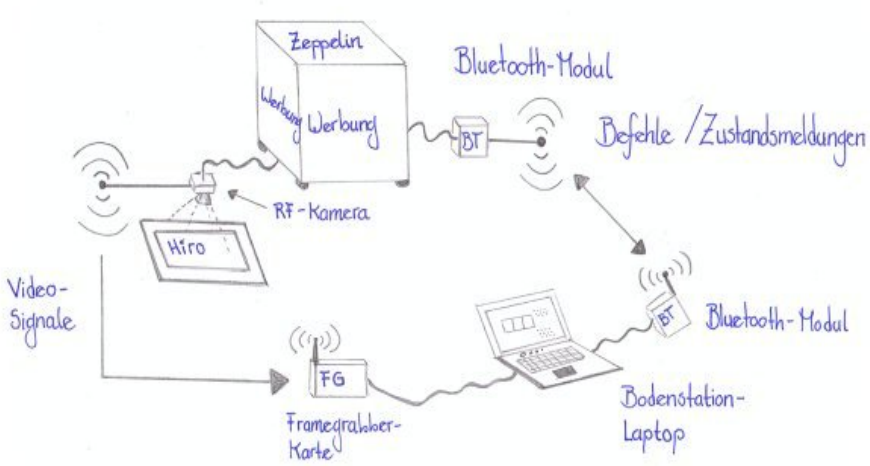


Abbildung 5.9.: Zeppelin: Zeichnung des Aufbaus

5.2. Software Aufbau eines Zeppelins

Die im Rahmen dieser Bachelorarbeit erstellte Software, wird im Teil „Realisierung “ im Kapitel 8.3 näher beschrieben.

Im Vorfeld sei darauf hingewiesen, dass bei der Durchführung der Tests Einschränkungen vorlagen. Dadurch konnte der eigentliche Programmieranteil auf einer Ersatzplattform erst später vorgenommen werden.

An dem in dem oben erwähnten Programmieranteil ändert sich durch das Portieren auf eine andere Plattform nur geringfügig etwas. Im Kapitel 8.3 wird ebenfalls auf diese Änderungen eingegangen.

5.3. Leistungsbeschreibung

Nach diesen Analysen ergibt sich für unseren geplanten Zeppelin ein fester Leistungsumfang.

5.3.1. Flugeigenschaften

Mit Helium als Trägergas soll ein ruhiger, störungsfreier Flug ermöglicht werden. Die Bauform wird jedoch nur einen reinen Indoor-Betrieb ermöglichen.

Durch die geplante Kantenlänge von 1,30 m können etwa 2,3 kg transportiert werden, so dass neben der reinen Konstruktion und den benötigten elektrischen Teilen noch Platz für weitere technische Komponenten ist.

5.3.2. Antrieb

Der Antrieb erfolgt über die beiden 80 mm Lüfter. Über spezielle Servomotoren sind diese schwenkbar und ermöglichen eine präzise Steuerung des Zeppelins. Die Stromversorgung wird über mitgeführte Li-Polymer-Akkumulatoren geschehen, so dass die größtmögliche autonome Flugdauer erreicht wird.

5.3.3. Steuerung

Die Grundsteuerung übernimmt das an Bord befindliche Aksenboard. Es wertet die Sensordaten aus und regelt entsprechend die Motorensteuerung. Die Kontrolle über die Erfüllung von Aufgaben übernimmt die Bodenstation. Über eine an Bord mitgeführte Funkkamera werden kontinuierlich Bilder an die Bodenstation gesandt und über eine Bilderkennungssoftware ausgewertet. Anhand dieser Auswertung wird eine Reaktion berechnet, die dann in Form eines konkreten Fahrbefehls an den Zeppelin geschickt wird.

5.3.4. Sensorik

An Bord befinden sich sechs Ultraschall-Sensoren, die zur Abstandsmessung dienen. An jeder Seite des Zeppelins ist einer angebracht. Die Sensoren übertragen ihre Daten via I²C-Bus an das Aksenboard. Hiermit wird erst ein autonomes Fliegen ermöglicht, da der Zeppelin so eigenständig Hindernisse erkennen und Ausweichmanöver berechnen kann.

5.3.5. Peripherie

Es wird eine Funkkamera mitgeführt, die Bilder per Funk an die Bodenstation senden kann. Ein Bluetooth Seriell Modul dient zur Kommunikation zwischen dem Aksenboard und der Bodenstation. Mit dieser Komponente könnte das Aksenboard während des Fluges neu geflasht werden und so auf veränderte Umgebungen reagieren. Dies könnte z.B. nötig werden, wenn sich die Umgebung stark ändert und die Sensoren angepasst werden müssen. Auch wäre der Fall möglich, dass der Zeppelin für den Verantwortlichen aufgrund einer Fehlfunktion nicht mehr greifbar ist. Statt eines mehr oder weniger kontrollierten Absturzes, könnte man so das Aksenboard neu flashen und die Fehlfunktion korrigieren.

5.3.6. Ziel

Das Ziel des Projektes mit dem Zeppelins ist, in Messen oder bei größeren Veranstaltungen gezielt Werbung einzusetzen. Basis hierbei ist das „autonome Fliegen ohne Anecken“. Der Zeppelin kann alleine vollautonom durch den Raum fliegen ohne an Hindernisse zu stoßen. Zusätzlich wird er sich an Pattern orientieren können und diese mit Hilfe der Bodenstation korrekt interpretieren können. So ist eine feste Route über ein Messegelände möglich oder auch ein Kreisen über einem bestimmten Messestand.

Teil III.

Design

6. ARToolkit

6.1. Generelle Einführung in das ARToolkit unter Bezugnahme auf das Projekt

Das ARToolkit ist ein in in der Programmiersprache C geschriebenes Framework, das über mehrere Schnittstellen Zugriff auf digitalisierte Bilder einer angeschlossenen Videokamera bietet. Kernfunktion ist eine Patternerkennung, die beliebige Pattern erkennen und auf diesem eine virtuelle Projektion erzeugen kann. Das aktuelle Kamerabild ist über ein Steuerpanel einsehbar.

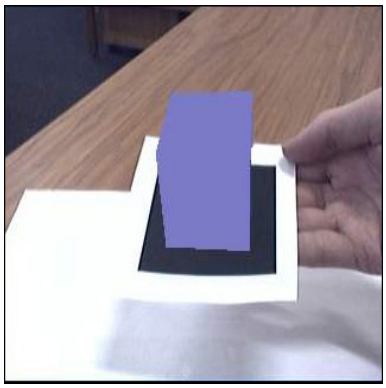


Abbildung 6.1.: Erkanntes Pattern mit Projizierung

6.1.1. Patterns

Ein Pattern ist ein Bildmuster, das bestimmten Anforderungen genügt und im Grundaufbau bei allen Pattern gleich ist.

Er besteht aus einem schwarzen Rahmen, in dessen Mitte eine quadratische Fläche für das eigentliche Muster zur Verfügung steht. Diese Fläche umfasst genau ein Drittel der schwarzen Umrandung und darf sich nur weiß absetzen. Dadurch entsteht ein maximaler Kontrast.

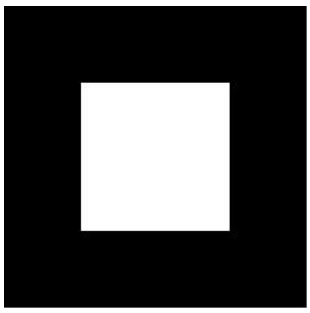


Abbildung 6.2.: Grundlayout eines ARToolkit Pattern

Auf diese weiße Fläche können beliebige Figuren oder Texte getaggt werden. Farbige Formen sind möglich, jedoch nicht empfehlenswert, da nur ein maximaler Kontrast wie Schwarz zu Weiß zur optimalen Erkennung führt. Folgende Beispielpattern sind im ARToolkit enthalten:

Abbildung 6.3.: Hiro Pattern

Abbildung 6.4.: Kanji Pattern

6.1.2. Funktionsweise

Die Framegrabberkarte liefert kontinuierlich Bilder von der Webcam, die vom ARToolkit verarbeitet werden. Aus einem Frame berechnet das Programm sofort eine Matrix. Diese wird intern in einem Array abgebildet und mit den Matrizen der bekannten Patterns verglichen. Im Falle einer Übereinstimmung wird sofort ein beliebiges virtuelles Objekt, wie z.B. ein Würfel, auf das Pattern im aktuellen Kamerabild projiziert.

Ab diesem Zeitpunkt steht die berechnete Matrix als Array zur Verfügung, mit dessen Hilfe unter anderem die relative Position der Kamera zum Pattern bestimmt wird.

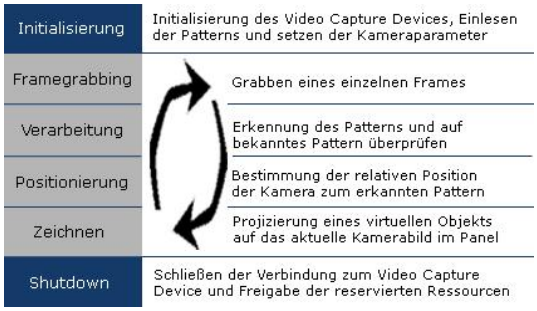


Abbildung 6.5.: Ablauf einer Patternerkennung

6.1.3. Verwendung in unserer Arbeit

In diesem Projekt wurde das ARToolkit nur zur reinen Bildmustererkennung verwendet. Die Funktionen zur Projektion von virtuellen Objekten auf das aktuelle Kamerabild fand ihren Nutzen, um visuell das Erkennen eines Patterns zu signalisieren. Für das Steuerpanel des Zeppelins wurde über Schnittstellen die Kernfunktionen des ARToolkits zugänglich gemacht.

Jeder Frame, den die Webcam liefert, wird über die generierte Matrix mit den vorliegenden Matrizen bekannter Patterns verglichen. Erfolgt auf einem Frame die Identifikation eines bekannten Pattern, wird ein Event ausgelöst. Dieses Event wird von der Steuerungssoftware getriggert und verarbeitet. Die Matrix und der Name des Patterns werden vom ARToolkit an die Steuerungssoftware weitergereicht und mit der „Properties“-Datei verglichen. Sind in der „Properties“-Datei Aktionen zu einem Pattern hinterlegt, wird die Aktion ausgeführt.

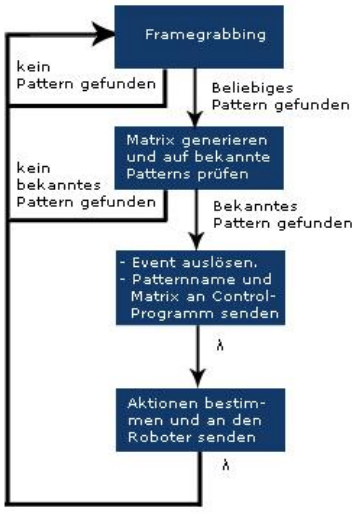


Abbildung 6.6.: Kombinierter Ablauf von ARToolkit und Zeppelin Steuerungssoftware

6.2. Installation und Handhabung

6.2.1. Voraussetzungen

Damit die freie Bibliothek ARTookit funktioniert, bedarf es wichtiger Voraussetzungen, die hier näher erläutert werden.

Wichtig ist die Tatsache, dass der Computer, auf dem die Bibliothek laufen soll, eine Grafikkarte besitzen muss, die entweder OpenGL oder Direct3D unbedingt unterstützt. Die angeschlossene Kamera sollte eine Framerate von 30 Frames pro Sekunde unterstützen. Darüberhinaus ist es von Vorteil, wenn die Auflösung der Kamera überdurchschnittlich gut ist, da dies einen direkten Einfluss auf das Erkennen der Pattern hat. Ein schneller Prozessor sowie genügend RAM sind ebenfalls wichtig, da die Ressourcen des Rechners durch die Echtzeiterkennung stark beansprucht werden.

Sollte man sich entschließen, das in den nächsten Kapiteln erwähnte JARToolkit zu benutzen, sollte man darauf achten, dass die Systemvoraussetzungen, für das in dem Fall ebenfalls benötigte Java3D Paket, gegeben sind.

Die besondere Herausforderung der gesamten Installation und des Zusammenspiels aller Einzelkomponenten liegt in der Beschaffung der richtigen Versionen der Komponenten. Es muss sehr genau darauf geachtet werden, welche Softwareversionen nötig sind, um die erforderliche Kompatibilität nach der Installation zu erreichen. Hier sollte man sich an den beiliegenden, zum großen Teil unvollständigen und schwer verständlichen Anleitungen orientieren, die die notwendigen Softwareversionen beschreiben, um möglichen Problemen bei der Verknüpfung vorzubeugen.

6.2.2. Installation

Installieren des ARToolkits

Bei der Installation des ARToolkit handelt es sich um eine aufwendige Prozedur. Die Bibliothek ist nur als nicht kompiliertes Projekt zum Download im Internet verfügbar. Binaries gibt es nicht. In der dem Projekt beiliegenden „Readme.txt„-Datei befindet sich eine kurze Anleitung, welche Tools man im Vorfeld benötigt, wie damit das ARToolkit kompiliert wird und startet bzw. benutzt werden kann. Im Anhang befindet sich eine in die deutsche Sprache übersetzte Anleitung, basierend auf die beiliegenden Instruktionen aus dem Englischen.

Um das ARTookit zu installieren, benötigt man eine integrierte Entwicklungsumgebung (hier: Microsoft Visual Studio). Dazu sei erwähnt, dass das ARToolkit eine Projektdatei für die

Entwicklungsumgebung „Visual Studio“ von der Firma Microsoft bereits beinhaltet und das Importieren dadurch sehr vereinfacht wird.

Es werden zwei weitere Bibliotheken für das Projizieren von dreidimensionalen Objekten auf das laufende gerenderte Videobild benötigt. Die erste DSVideoLib ist für die Videokamera und stellt die generelle Schnittstelle zu dem speziellen Videotreiber der Kamera dar. Die zweite namens GLUT ist eine plattformunabhängige Bibliothek für das Schreiben von OpenGL Anwendungen. Im Rahmen des ARToolkit wird sie zum „Zeichnen“ der 3D-Objekte benutzt.

Sobald die Entwicklungsumgebung installiert ist und die beiden benötigten Bibliotheken richtig eingebunden sind, kann die Visual Studio Projektdatei geöffnet und das Projekt komplett kompiliert werden.

Als Resultat entsteht eine „artoolkit.dll“-Bibliothek sowie mehrere ausführbare Testprogramme, die einen ersten Eindruck über den Funktionsumfang erlauben. Die im PDF-Format beiliegenden Pattern sollten zur Durchführung von Tests als Ausdruck vorliegen. Für den optimalen Verlauf muss die Kamera im Vorfeld an den Rechner angeschlossen sein. Eine detaillierte Anleitung in der deutsch Sprache für das Installieren aller benötigten Komponenten findet sich im Anhang wieder.

Das JARToolkit

Es existiert ein auf das ARToolkit aufbauendes Projekt namens JARToolkit, eine Bibliothek, die in Java geschrieben wurde. Diese Bibliothek erlaubt es, Programme in Java zu erstellen, die dann wiederum über Native-Schnittstellen auf das in „C“ geschriebene ARToolkit zugreifen. Das JARToolkit wird scheinbar nicht mehr weiterentwickelt, und beinhaltet noch einige Fehler beim Erkennen von Multiplen Pattern. Es bietet aber eine hervorragende Funktionsweise. Leider gibt es zu der in Java geschriebenen Bibliothek kaum eine Dokumentation. Es ist nötig, sich autodidaktisch das dazugehörige Wissen anzueignen. Auch für dieses Projekt existiert eine Installationsanleitung, die ebenfalls in die deutsche Sprache übersetzt wurde und in dem Anhang genannt wird.

Das JARToolkit wird dann in den Klassenpfad des zu erstellenden Javaprojektes eingebunden. Im Rahmen dieser Arbeit wird die Java Entwicklungsumgebung Eclipse benutzt. Dort kann das JARToolkit direkt eingebunden werden. Zum Zeichnen der 3D-Objekte benötigt man, zusätzlich zu dem Java SDK, noch die Java3D API. Hierbei muss besonders auf die SDK Version geachtet werden, da Java3D nicht mit jeder SDK richtig ausgeführt wird.

Sobald die Installation erfolgreich abgeschlossen wurde, kann mit dem Programmierteil begonnen werden.

6.2.3. Probleme und wichtige Hinweise

Probleme entstehen eigentlich nur dann in Verbindung mit dem (J)ARToolkit, wenn die Vorbedingungen nicht eingehalten werden und der Rechner sowie die Kamera nicht die Hardwarevoraussetzungen erfüllen.

Wichtig ist, wie bereits erwähnt wurde, eine qualitativ hochwertige Kamera mit hoher Auflösung und hoher Bildwiederholungsrate sowie eine leistungsstarker Rechner, der genügend Ressourcen aufweist, um Programme in Echtzeit zu bewältigen. Das Betriebssystem ist eher zweitrangig, da das ARToolkit sowohl auf einem Windows- wie auch auf einem Linux- oder Mac-System kompiliert werden kann. Durch die Java Schnittstelle JARToolkit werden die Programme vollends plattformunabhängig.

Um Javaprogramme zu starten, die sowohl das JARToolkit als auch Direct3D zum Rendern der 3D-Objekte benutzen, muss beim Starten der Java Virtuellen Maschine mit dem Befehl:

java **-Dj3d.rend=d3d** <Programm-Name>

mitgeteilt werden, dass das Programm nicht die standardmäßig definierte OpenGL Schnittstelle gebrauchen soll sondern Direct3D.

7. Probleme bei der Realisierung des Zeppelins

Bereits bei der Verarbeitung der einzelnen Komponenten zeigten sich teilweise erhebliche Probleme, welche auch zu leichten Bedenken bezüglich der Flugtauglichkeit führten. Durch die Zusage der Vollführung einer Präsentation bei der „Langen Nacht des Wissens“, schmälerte sich der zeitliche Rahmen erheblich.

7.1. Organisatorische Probleme

7.1.1. Besorgung des Materials

Aufgrund der Vielzahl an verschiedenen Teilen, die für das gesamte Projekt benötigt wurden und des beschränkten Budgets, erwies sich das Beschaffen der einzelnen Komponenten als sehr aufwendig. Bei diversen Internet-Fachhändlern wurden Bestellungen aufgeben und die Assistenten im Labor für Softwaretechnik mussten um die Besorgung diverser Verbrauchsmaterialien bemüht werden.

Einige Komponenten trafen erst kurzfristig vor der ersten Präsentation ein, so dass für Testdurchläufe kaum Zeit blieb.

Das größte Problem war hierbei die Beschaffung und sachgerechte Lagerung des zum Auftrieb benötigten Heliums. Die 10 m³ Flasche Helium kostet rund 180 EUR und muss extra über das „Institut für Werkstoffkunde “ bestellt werden. Diese riesige etwa 60 kg schwere Gasflasche konnte aufgrund der Sicherheitsbestimmungen nur im Keller der HAW gelagert werden. Aus sicherheitstechnischen Gründen konnte die Befüllung mit Helium daher nur im Keller der HAW vorgenommen werden.

7.1.2. Nutzung von Werkstätten

Die Arbeiten und Techniken, die für die Durchführung des Projektes durchgeführt werden mussten, waren vielfältig. Entsprechend umfangreich war die Palette an den Werkzeugen, mit denen die Arbeit verrichtet wurde. Es galt, in mehreren Fachbereichen Termine in Werkstätten auszumachen; die Kooperation mit der Werkstattleitung war unumgänglich. Nur mit einem hohen Maß an Überzeugungsarbeit konnten zur ersten Präsentation alle Arbeiten in den Werkstätten abgeschlossen werden.

7.2. Handwerkliche Probleme

Die Arbeit war nicht nur auf den Bereich der Informatik beschränkt, sondern umfasste gerade im Anfangsstadium handwerkliche Arbeiten, für die man als angehender Informatiker nicht vorbereitet ist. Trotz gewissenhafter Einarbeitung gab es hier eine hohe Fehlerquote.

7.2.1. Verarbeitung des Grundgerüsts

Die Bearbeitung der Werkstücke nach einem selbst erstellten Konstruktionsplan erwies sich als eine herausfordernde Aufgabe. Trotz vieler und zum Teil grundlegend verschiedener Teile, waren die Hauptkomponenten solide verarbeitet und fehlerfrei in ihrem Ablauf. Das einziges Problem lag bei der fehlenden Flexibilität der Bauteile, die das Testen enorm erschwerten. Die Teile mussten fest miteinander verankert sein, um höchste Stabilität zu gewährleisten. Der starre Aufbau führte bei dieser Größe zu Transport- und Lagerungsproblemen.

7.2.2. Schweißen der Ballonhülle

Die wichtigste und auch fehleranfälligste Komponente bei einem Zeppelin ist die Außenhülle. Es besteht immer die Gefahr, dass sie einreißt oder undicht ist. Als Gegenmaßnahme müsste man ständig die Heliumflasche mitführen und das entweichende Gas nachfüllen, was bei der Größe des Zeppelins undenkbar ist.

Der sichtbare Erfolg dieses Projektes hängt direkt von der Qualität der Außenfülle ab.

Die Folie, die für die Außenhülle verwendet wurde, konnte nur in 0.5 m x 7 m Streifen erworben werden. Dadurch war es unvermeidlich, alles in kleinere Stücke zu zerschneiden und anschließend gemäß der Vorberechnungen wieder zusammenzuschweißen. Dies erwies sich im nachhinein als ein gravierender Fehler. Es musste eine insgesamt 30 m lange

Schweißnaht sauber verarbeitet werden. Das Werkzeug, das dafür zur Verfügung stand, war eine 10 cm breite Schweißzange.

Da keine Alternativen zur Auswahl standen, musste damit gearbeitet werden, auch wenn es von vorneherein als schlechte Wahl erschien.

Zur Sicherheit wurden vor Inbetriebnahme diverse Tests mit Druckluft durchgeführt, um die Dichte der Folie sicherzustellen. Dabei konnten keine größeren Löcher mehr ausmacht werden.

7.2.3. Räumlichkeiten

Die Ansprüche an die Räumlichkeiten eines Zeppelins mit einer Kantenlänge von 1,3 m sind sehr hoch. Es muss genügend Platz für den Zeppelin selbst und das Arbeiten an dem Modell vorhanden sein. Hinzu kommen eine ständige Spannungsquelle und ein Arbeitstisch.

Die Akkus des Zeppelins müssen ständig wieder aufgeladen werden. Beim plötzlichen Auftretens eines Fehlers muss man in der Lage sein, direkt mit dem Messgerät Probemessungen durchführen zu können.

Leider gab es im Fachbereich E/I an der HAW keine geeignete Räumlichkeit, die den Anforderungen genügte. Die für uns zugängliche Werkstatt war im 11. Stock nur über den Fahrstuhl erreichbar genau wie das Labor für „Robotics“. Beide hatten zu schmale Türen und nicht genügend Raum zum Testen. Der einmal voll aufgeblasene Zeppelin wäre mit seiner Kantenlänge nicht mehr aus dem Stockwerk herausgekommen. Zudem gab es Schwierigkeiten bei der Lagerung der Heliumflasche. So waren die Testmöglichkeiten sehr beschränkt und die Flugtests konnten gezwungenermaßen nur im Keller des Fachbereichs durchgeführt werden.

7.3. Die 1. „Lange Nacht des Wissens“

Passend zu diesem Ereignis sollte der Jungfernflug vor Ort stattfinden. Nachdem alle Komponenten einzeln fertig gestellt worden waren, konnte rechtzeitig damit begonnen werden, alles mit einander zu verbinden. Leider waren die Komponenten nicht so flexibel und steckbar, wie zuerst erhofft.

Da das Befüllen des Zeppelins im Keller der Hochschule vorgenommen werden mußte, gestaltete sich der Transport des aufgeblasenen Zeppelins in die Ausstellungshalle als schwierig. Im Nachhinein ist es schwierig festzustellen, woran es lag, aber wahrscheinlich sind dabei ein Teil des Aksenboards sowie die Ballonhülle beschädigt worden.

Auf der Veranstaltung konnte zwar der Jungfernflug planmäßig gestartet werden, jedoch gestaltete sich der Flug als recht kurz. Durch einige nicht feststellbare Risse in der Folie und einer wahrscheinlich durch den Transport beschädigten H-Brücke am Aksenboard, war nur ein kontrolliertes Schweben möglich.

Am Ende des Abends war fast die Hälfte der Heliumflasche aufgebraucht. Da der aufgeblasene Zeppelin aufgrund seiner Größe nicht mehr aus der Ausstellungshalle transportieren werden konnte, musste ein Großteil an Helium abgelassen werden. Alles in Allem wurden an diesem Abend knapp 5 m^3 Helium verbraucht, was einen Kostenpunkt von 90 EUR ausmacht.

Daher werden wir uns im nächsten Kapitel mit der Implementierung einer Ersatzplattform beschäftigen und hardwareseitig einen Umbruch bei gleich bleibender Software durchführen.

Teil IV.

Realisierung

8. Portierung der Grundidee auf eine umsetzbare Plattform

Durch die Erkenntnisse der ersten Umsetzungsphase und die in Kapitel 7 beschriebenen Probleme, wurde die weitere Umsetzung prototypisch fortgesetzt. Die Idee wurde vom dreidimensionalen in den zweidimensionalen Raum portiert.

Statt eines Zeppelins, der über Besucher von Messenhallen fliegt, gibt es nun einen Roboter mit dem Aksenboard. Dieser fährt auf dem Boden durch Räume und erkennt mit einer kleinen Kamera Patterns.

Die Patterns sind an der Wand angebracht und werden so direkt vom Roboter erkannt. Abstrahiert betrachtet, wurde die ursprüngliche Problemstellung nur leicht abgeändert. Anstatt dass ein Zeppelin von oben Marken erkennt und entsprechend reagiert, fährt nun ein Roboter durch Räume, erkennt Marken und reagiert auf diese. Da der Preis für einen Flug des Zeppelins für das universitäre Umfeld viel zu teuer ist, erwies sich der Verzicht auf diese Flugfähigkeit als sinnvoll.

Ein weiterer Vorteil der Portierung ist, dass die neue Plattform ein Subnotebook aufgeschnallt bekommen kann. So hat man ein vollständig abgeschlossenes System und benötigt im Gegensatz zum ursprünglichen Zeppelin keine Bodenstation mehr.

8.1. Entwurf einer neuen Plattform

Bei genauerer Betrachtung wurde die ursprüngliche Idee nur leicht abgeändert. Der äußere Aufbau ist zwar komplett unterschiedlich, aber die Kernkomponenten sind absolut identisch. Es wurde erneut das Aksenboard mit den bekannten Sharpsensoren verwendet. Auch die Kamera und das Bluetooth-Seriell-Modul haben weitere Verwendung gefunden.

Die gesamte Software, die benutzt wurde, ist identisch mit der des Zeppelins. Gleiches gilt für den C-Code des Aksenboards, der mit wenigen Änderungen übernommen wurde.

8.1.1. Features

Grundsätzlich sollten sich die Eigenschaften der neuen Plattform nicht von denen des Zeppelins unterscheiden. Im Gegenteil: Dadurch, dass nun nicht mehr auf das Gewicht geachtet werden mußte, konnte mit den Ressourcen großzügiger umgegangen werden.

Vollautonomes Fahrverhalten

Das größte Feature ist sicherlich, dass die neue Plattform ein Subnotebook ergänzend erhalten hat. So hat man ein abgeschlossenes, vollautonomes System und benötigt im Gegensatz zum ursprünglichen Zeppelin keine Bodenstation mehr. Auf diesem Laptop läuft die komplette Software, um den Roboter zu steuern. Wenn man diesen Rechner über WLAN fernsteuern würde, könnte man sogar während des Betriebes das Aksenboard flashen und mit der Steuerungssoftware fernsteuern.

Sounds

Der Roboter zeigt jetzt an, dass er ein Pattern gefunden hat, in dem er einen voreingestellten Sound mit einem eigenen Audioplayer wiedergibt. Aufgrund der vielen nebenläufigen Prozesse der Software, wurde eine spezielle Queue entwickelt. Die Patternsuche läuft in einer Endlosschleife ab, und es könnten neue Pattern erkannt werden, obwohl bereits ein Sound abgespielt wird. Die Queue stellt sicher, dass das Abspielen der Sounds in der richtigen Reihenfolge geschieht.

Optimierungen am Aksenboard

Bei der Arbeit mit einem reaktiven System, speziell wenn Daten zwischen Teilsystemen übertragen werden, ist es wichtig, die laufenden Prozesse zu synchronisieren. Das setzt voraus, dass alle Teilsysteme Threads unterstützen. Was bei Java schon implementiert ist, gestaltet sich beim Aksenboard schwierig. Die Implementierung ist noch nicht ausgereift und wahrscheinlich für den Mikrocontroller auch überdimensioniert. Es waren aber unbedingt Nebenläufigkeiten unabdingbar, da der Roboter autonom fahren und gleichzeitig auf einen Befehl am seriellen Port warten sollte. Das Warten auf serielle Eingaben ist beim Aksenboard über ein blockierendes Lesen gelöst. Dieses hätte die gesamte Funktionalität gestört.

Daher wurde über ein Upgrade der Fachhochschule Brandenburg die Firmware unseres Aksenboards aktualisiert und zumindest eine rudimentäre Unterstützung für Nebenläufigkeiten erlangt.

Das Synchronisieren der Prozesse war ein zweites, schwerwiegendes Problem. Das Aksenboard sendet Zustandsmeldungen und empfängt Befehle über den seriellen Port. Dieser bietet von sich aus kein serielles Handshaking.

Folgende Situation kann schon zu einem Deadlock führen:

- Ein Pattern wurde erkannt.
- Die Steuerungssoftware findet eine passende Aktion in der „Properties“-Datei und sendet einen Fahrbefehl zum Aksenboard.
- Das Aksenboard empfängt den Befehl und führt ihn aus.

Nun hat die Steuerungssoftware aber keinerlei Kenntnis darüber, ob der Fahrbefehl angekommen ist und ausgeführt wurde. Da die Patternsuche zyklisch abläuft, würden jetzt weitere Patterns gesucht und weitere Fahrbefehle an das Aksenboard gesandt. Das ganze System wäre asynchron, da die Steuerungssoftware nicht mehr den Zustand des Aksenboards kennt.

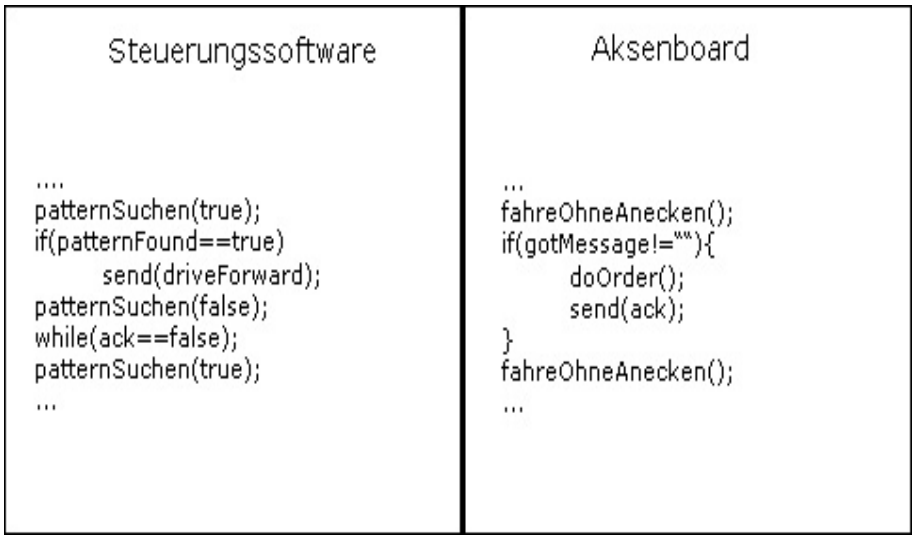


Abbildung 8.1.: Serielles Handshaking

Dazu haben wir ein Handshaking entwickelt. Dies ähnelt dem TCP/IP Protokoll Stack. Kern ist, dass Patterns nur dann überhaupt gesucht werden, wenn das Aksenboard keinen aktuellen Fahrbefehl bearbeitet.

8.1.2. Beispiel Mission

Als Mission bezeichnen wir den Ablauf eines möglichen Szenarios. Es gibt einen groben Ablauf, der fix ist und für alle Missionen identisch aufgebaut ist. Dieser gleicht genau dem ursprünglich für den Zeppelin vorgesehenem Ablauf.

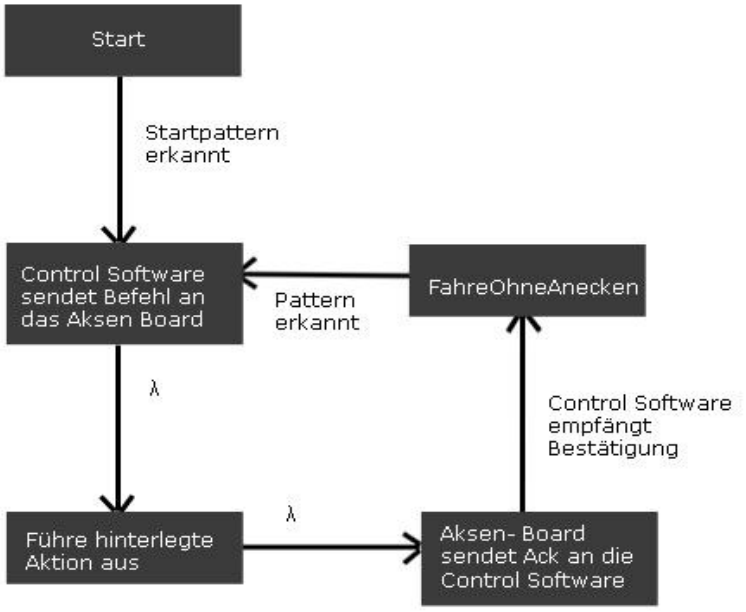


Abbildung 8.2.: Zustandsautomat des Gesamten Systems

Der Roboter wird in Betrieb genommen woraufhin er bereit für Anweisungen ist. Jetzt wird ihm Pattern A gezeigt, woraufhin er losfährt. Er fährt solange im Raum, natürlich ohne irgendwo gegen zustoßen, bis er ein Pattern B, C oder D findet.

Sobald er eines davon findet, stoppt er auf der Stelle und signalisiert akustisch, welches Pattern er gefunden hat und welche Aktion für dieses Pattern in der „Properties“-Datei hinterlegt ist. Nachdem der Roboter über die Patterns eine beliebige Route abgefahren hat, geht er hier in den Stand-By Modus und wartet erneut auf das Startpattern.

8.1.3. Vergleich zwischen alter und neuer Plattform

Große Unterschiede zur ursprünglichen Plattform gibt es lediglich bei der Hardware. Der Aufbau ist anders geartet und auch die Dimensionen unterscheiden sich. Betrachtet man aber die Software und die finalen Funktionalitäten, so fällt der Unterschied sehr gering aus.

Wir waren angetreten, um einen Zeppelin in Messehallen teilautonom fliegen zu lassen und diesen anhand von noch nicht näher definierten Wegmarken über den Köpfen der Besucher als schwebende Werbefläche anzubieten. Dies war aus Kosten- und Materialgründen nicht umsetzbar. Nun wurde ein Roboter konstruiert, der auf dem Boden vollautonom umherfährt und anhand von fest definierten Patterns seine Mission ausführen kann. Es wurden, abgesehen vom Rohbau, alle Komponenten wie Sensorik, Controller und sogar die Software komplett vom Zeppelin übernommen.

Somit sind zwar äußerlich deutliche Unterschiede zwischen den beiden Plattformen vorhanden, aber der Grundgedanke bleibt der gleiche. Für unsere Software ist es kein Unterschied, ob Sie einem 12 V Motor am Roboter oder einem Propeller am Zeppelin den Fahrbefehl erteilt. Gleichermaßen funktioniert die Patternerkennug am Boden genauso wie in der Luft. Durch die Portierung konnten wir sogar noch einige Features mehr einsetzen, die das Projekt noch attraktiver gestalten. Als Beispiel sei hier zum Beispiel auf die Audioausgabe verwiesen, die ohne aufgeschnalltes Notebook nicht möglich gewesen wären.

Wenn sich auch die äußere Erscheinung des Projektes gewandelt hat, so bleibt die Idee, vor allem aber die Theorie und technische Umsetzung dahinter, kongruent. Aus den beschriebenen Schwierigkeiten mit dem Zeppelin wurde ein neuer Arbeitsrahmen erschlossen, in welchem mit den gleichen technischen Mitteln die Idee des autonomen, potentiellen Werbeträgers in der Form eines Roboters wieder aufgelebt werden konnte. Aus der Not wurde nicht eine Tugend, sondern ein noch viel versprechenderes Projekt.

8.2. Hardwareaufbau

Der neue Roboter basiert auf der mechanischen Plattform von Frank Freyer, eines Komillitonen, und einer modifizierten Version des Aksenboards.

Das Grundgerüst der neuen Plattform ist eine solide Metallkonstruktion von Frank Freyer. Sie besteht in Ihrer Rohform aus zwei übereinander liegenden Metallplatten. Unterhalb der Platten befinden sich die Motoraufhängung und die beiden Motoren.

Abbildung 8.3.: Aksenboard der FH Brandenburg mit modifizierten Motorport

Abbildung 8.4.: Geschmiedete Plattform von Frank Freyer

Abbildung 8.5.: Plattform mit Laptop

8.2.1. OnBoard Laptop

Auf Grund seiner Größe passt das JVC MP-XV841DE Subnotebook, welches im Rahmen dieser Arbeit zu Verfügung stand, optimal auf die obere Plattform des neuen Roboters. Mit einer Akkulaufzeit von über 4 Stunden ist der gesamte Roboter nahezu unabhängig von Steckdosen.

Da das Subnotebook über keinen seriellen Port zur Kommunikation mit dem Aksenboard verfügt, wurde stattdessen ein USB-Seriell-Adapter verwendet. Auf dem JVC läuft das Java Steuerungsprogramm mit der dazu gehörigen Oberfläche und die Software zum Flashen des Aksenboards. Der Laptop besitzt eine Direct3D-fähige Grafikkarte, welche Vorraussetzung für JARToolkit ist.

8.2.2. Antrieb

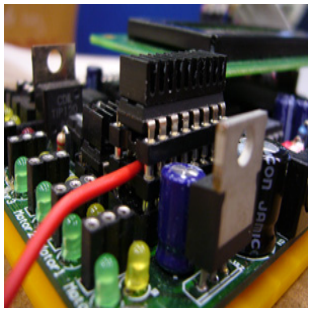


Abbildung 8.6.: Aksenboard: Haarbrücke

Die neue Plattform hatte als Antrieb zwei Motoren, die 12 V benötigen. Da das Aksenboard von Haus jedoch nur 5 V über die Motorports liefern kann, war eine Modifikation an den Motorports notwendig.

Hiermit ist es nun möglich, Motoren an das Aksenboard anzuschließen, die wesentlich mehr als die gewöhnlichen 5 V benötigen. Man muss nur darauf achten, dass die angeschlossenen Motoren nicht mehr als 1 A benötigen, da die Haarbrücken nicht mehr Strom liefern können. Die Vorteile sind offensichtlich, der Antrieb wird von der Stromversorgung des Aksenboards entkoppelt, so dass in Zukunft stärkere Motoren nutzbar sind. Mit diesen können dann schwerere Lasten transportiert oder unwegsameres Gelände befahren werden.

8.2.3. Stromversorgung

Die Stromversorgung geschieht über einen 9,6 V Blockakku. Dieser etwa 10 cm lange Akku liegt unterhalb der Plattform und kann über einen selbstgebauten Schalter komplett vom System getrennt werden. Die Kapazität des Akkus ermöglicht eine Betriebsdauer von etwa 30 min.

Da das Aksenboard lediglich eine Spannung von max. 8 V verträgt, die Motoren und die Sharp-Entfernungssensoren allerdings 9,6 V benötigen, mussten wir einen Spannungsverteiler selber herstellen, der beide Spannungen mit den dazu passenden Schnittstellen implementiert. Darüber hinaus sollte der Verteiler einen An/Aus-Schalter haben, über den man die Stromzufuhr schnell abstellen kann.

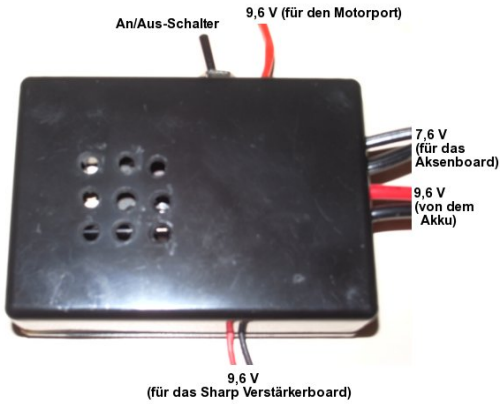


Abbildung 8.7.: Spannungsverteiler

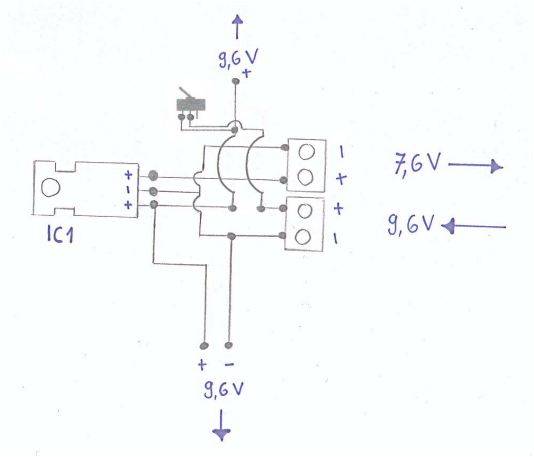


Abbildung 8.8.: Schaltplan Verteiler

Die Sharpsensoren benötigen darüber hinaus ein Verstärkerboard, eine Eigenentwicklung der Hochschule, welches eine Spannungsversorgung von 9,6 V an die Sensoren sowie die erfassten Sensorwerte gebündelt an das Aksenboard weiterleitet. Das Verbindungskabel zwischen Verstärker- und Aksenboard ist ein 14-poliges Flachbandkabel, welches eine ganz bestimmte Belegung besitzt. Bisher war es nur möglich die richtige Belegung durch das Vergleichen mehrerer Vorlagenbilder herzustellen. Nachfolgend ist eine übersichtliche Pinbelegung:

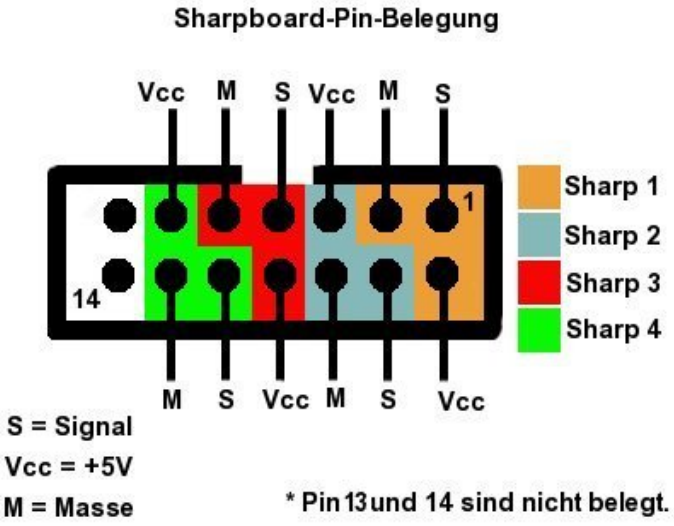


Abbildung 8.9.: Pinbelegung des Sharp-Verbindungskabels

8.2.4. Peripherie und Sensoren

Am Roboter sind insgesamt vier Sharp Sensoren angebracht.

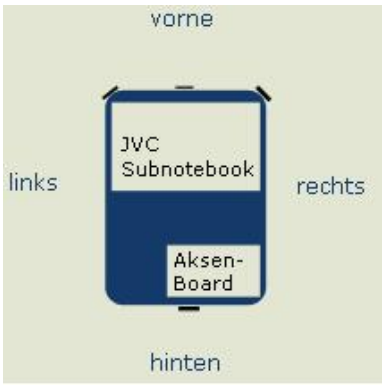


Abbildung 8.10.: Position der Sharps

Abbildung 8.11.: Plattform mit Sharps

Drei Sensoren befinden sich vorne und können etwa 180° des vorderen Blickfeldes abdecken. Der hintere Sensor kann für rückwärtiges Fahren verwendet werden.

An dem Laptop mitangeschlossen ist eine einfache USB-Kamera. Diese liefert das Bildmaterial für das ARToolkit.

8.3. Softwareaufbau der Plattform

Die Software der Plattform unterteilt sich in zwei Teile: Erstens in die Plattformsoftware des Roboters und zweitens in die Steuerungssoftware, verantwortlich für die Erkennung der Patterns. Jeder der beiden Teile funktioniert auch unabhängig voneinander. Die Plattform kann mit der Steuerungssoftware fahren, allerdings ohne auf Patterns reagieren zu können, während die Erkennung der Patterns auch ohne eine Roboter Plattform und deren Steuerungssoftware funktioniert. Beide Teile zusammen ergeben das komplette Paket, damit die Plattform autonom fahren und auf Patterns reagieren kann.

8.3.1. Roboter Programmierung

Der Roboter fährt selbstständig mit den beiden Motoren und den Entfernungssensoren. Außerdem befindet sich das Aksenboard auf der Plattform, welches die Steuerung übernimmt.

Der Mikrocontroller wird extern (auf einem Rechner) programmiert und das entstandene Programm wird über die serielle Schnittstelle auf das Board übertragen. Die Grundbibliotheken werden vorher auf das Board, ebenfalls über die serielle Schnittstelle, übertragen. Durch eine vordefinierte Bibliothek ist die Programmierung des Mikrocontrollers einfach.

Es wurde nach dem Subsumption-Model gearbeitet und abstrahierte Funktionen wie „Vorwärtsfahren“ oder „Drehung nach Rechts“ aus den elementaren Funktionen des Aksenboards erstellt. Dadurch können später einfachere Aktionen durch komplexere überlagert werden. Der spätere Quellcode wird deutlich schlanker. Da das Programm in einem einfachen C- Derivat geschrieben wird, sind die technischen Möglichkeiten begrenzt. Für die Implementierung von Nebenläufigkeiten musste ein Firmware Update durchgeführ werden.

„Fahren ohne Anecken“

Das Grundprogramm des Roboters ist eine für ihn lebenswichtige Funktion. Hierbei fährt der Roboter solange in einem für ihn fremden Raum, bis er ein Pattern findet und darauf reagieren muss bzw. ein Hindernis seine Weiterfahrt verhindert. Durch die beiden Motoren (und ein drittes Stützrad), die die Plattform besitzt, ist es ihm möglich jede zweidimensionale Bewegung zu vollführen oder Richtung anzufahren. Dadurch dass man die Motoren in beide Richtungen drehen und auch die Geschwindigkeit variabel einstellen kann, ist es dem Roboter möglich, sich sogar auf der eigenen Achse zu drehen.

Die Motorports des Aksenboards werden durch vordefinierte Funktionen angesteuert, die in der im Lieferumfang befindlichen Bibliothek sind.

Das Erkennen von Hindernissen wird durch Sharp-Entfernungssensoren geregelt. Bei Unterschreiten einer vordefinierten Entfernung zu einem Hindernis wird auf dieses Ereignis reagiert. Je nachdem welcher der vier Sensoren die kürzeste Distanz zum Hindernis verzeichnet, wird eine bestimmte Maßnahme eingeleitet. Vorne besitzt die Plattform drei Sensoren, da der Roboter hauptsächlich vorwärts fährt und dadurch eine genauere Bestimmung des Ausweichmanövers vorgenommen werden kann.

Die Sensoren benötigen ein „Verstärkerboard“, welches eine Eigenentwicklung der Hochschule ist. Es stand leider zu dem Zeitpunkt der Arbeit lediglich ein defektes Board zur Verfügung, bei dem einer der Schnittstellen für die Sensoren nicht richtig funktionierte. Softwaretechnisch wird der Sensor allerdings berücksichtigt. Wir haben uns entschlossen, den „Sensor 4“ dieser Schnittstelle zuzuordnen, da wir im Rahmen unserer Arbeit nicht zwingend auf Hindernisse hinter dem Roboter reagieren müssen. Als Ausgleich haben wir durch die vorderen drei Sensoren insgesamt eine ausreichend feine Ausweichstrategie.

Folgende Tabelle erfasst die verschiedenen Ausweichstrategien beim Erkennen eines Hindernisse durch einen (oder mehreren Sensoren):

Sensor	Position	Ausweichstrategie
Sensor1	Vorne-Links	• Rechter Motor wird auf halbe Kraft vorwärts eingestellt • Linker Motor wird auf volle Kraft vorwärts eingestellt • Beide Motoren werden auf volle Kraft vorwärts eingestellt Hierdurch fährt der Roboter eine sanfte Rechtskurve vorwärts. Nach insgesamt 2 Sek. fährt der Roboter wieder vorwärts.
Sensor2	Vorne-Mitte	• Motoren stoppen • Motoren fahren bei voller Kraft 1 Sek. rückwärts • Rechter Motor wird auf halbe Kraft rückwärts eingestellt • Linker Motor wird auf volle Kraft rückwärts eingestellt • Beide Motoren werden auf volle Kraft vorwärts eingestellt Hierdurch fährt der Roboter erst ein Stück zurück und dann fährt er eine leichte Rechtskurve weiterhin rückwärts. Nach insgesamt 2 Sek. fährt der Roboter wieder vorwärts.

Tabelle 8.1.: Ausweichstrategien des Roboters (Teil 1)

Sensor	Position	Ausweichstrategie
Sensor3	Vorne-Rechts	• Linker Motor wird auf halbe Kraft vorwärts eingestellt • Rechter Motor wird auf volle Kraft vorwärts eingestellt • Beide Motoren werden auf volle Kraft vorwärts eingestellt Hierdurch fährt der Roboter eine sanfte Linkskurve vorwärts. Nach insgesamt 2 Sek. fährt der Roboter wieder vorwärts.
Sensor4	Hinten	• Motoren stoppen • Motoren fahren bei voller Kraft 1 Sek. vorwärts • Rechter Motor wird auf halbe Kraft vorwärts eingestellt • Linker Motor wird auf volle Kraft vorwärts eingestellt • Beide Motoren werden auf volle Kraft vorwärts eingestellt Hierdurch fährt der Roboter erst ein Stück vorwärts und dann fährt er eine leichte Rechtskurve weiterhin vorwärts. Nach insgesamt 2 Sek. fährt der Roboter wieder vorwärts.

Tabelle 8.2.: Ausweichstrategien des Roboters (Teil 2)

Externe Befehle

Der Roboter hat die Möglichkeit, zur Laufzeit auf Ereignisse zu reagieren, die über die serielle Schnittstelle den Mikrocontroller erreichen.

Da das Lesen der seriellen Schnittstelle blockierend verarbeitet wird, müsste der Roboter solange stehen bleiben, bis ein Ereignis eintritt. Da wir allerdings „Fahren ohne Anecken“ und erst im Falle eines externen Befehles etwas anderes ausführen wollen, ist es unumgänglich, beide Funktionen gleichzeitig verrichten zu können. Daher wurde beschlossen, auf dem Aksenboard mit Threads zu arbeiten. Dies ist erst seit dem Update auf die letzte Version der Aksenbibliothek möglich, aber noch sehr fehleranfällig und musste über ein Firmware Update eingespielt werden.

Folgender Quellcodeausschnitt zeigt das Erzeugen beider benötigter Threads:

```
int main(void){
Thread1 = fahreOhneAnecken();
// Sicher durch den Raum fahren
// Ohne irgendwo hängen zu bleiben
Thread2 = serielleKommunikation();
// Auf serielle Eingaben warten
// Aktionen anstoßen und Bestätigungen senden
}
```

Zuerst wird Thread1 gestartet, der über Zugriffe auf Sensoren und Motoren ein fehlerfreies Fahren gewährleistet. Hierdurch wird garantiert, dass auch bei defekter serieller Kommunikation das System in einem sicheren, konsistenten Zustand bleibt. Dieser Thread läuft immer im Hintergrund, mit der Ausnahme, wenn das Aksenboard einen expliziten Befehl über die serielle Schnittstelle empfängt. Dann wird der Thread in den schlafenden Zustand verdrängt.

Thread2 läuft pseudoparallel zu Thread1. Er stellt die Kommunikation über die serielle Schnittstelle her und pollt auf eingehende Nachrichten. Beim Eintreffen einer Nachricht, wird Thread1 auf schlafend geschaltet und die Nachricht verarbeitet. Hiernach wird über die serielle Schnittstelle eine Bestätigung rausgeschickt. Die Synchronisierung der beiden Threads wird über einen Semaphor sichergestellt.

Der komplette, kommentierte Quellcode befindet sich im Anhang.

8.3.2. Patternerkennung und Steuerung

Die Patternerkennung wird, wie bereits im Kapitel 6 ausführlich beschrieben, über das ARToolkit und deren Java Schnittstelle realisiert. Wir haben im Rahmen dieser Arbeit ein Programm implementiert, welches sich die Fähigkeiten des ARToolkits zu Nutze macht und den Roboter über die serielle Schnittstelle steuert. Das Programm ist in Java geschrieben und dadurch auf jeder Plattform, für die es eine Java Virtuelle Maschine gibt, lauffähig.

Die Steuerungssoftware

Die Steuerungssoftware ist eine interaktive, audiovisuelle Schnittstelle für den Benutzer. Sie ermöglicht es dem Benutzer durch Tasten- und Maussteuerung den Roboter fernzusteuern. Man bekommt außerdem durch eine graphische Anzeige vermittelt, welche Bewegung der Roboter gerade ausführt, und durch eine weitere Anzeige, was der Roboter gerade durch seine Kamera sieht.

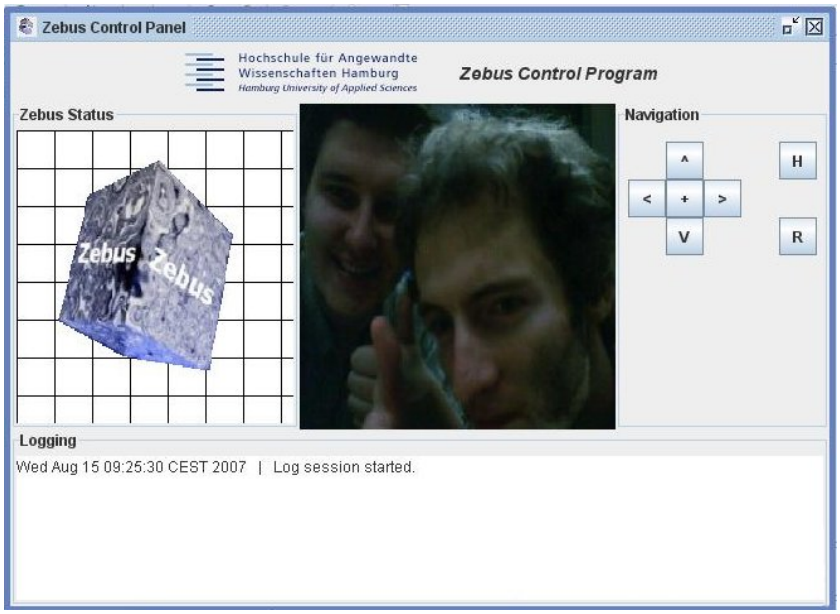


Abbildung 8.12.: Oberfläche der Steuerungssoftware

Des Weiteren besitzt die Software die Möglichkeit, Musik oder Sound jeglicher Art abzuspielen, welches durch eine selbst geschriebene Softwarebibliothek implementiert wird. Die Sounds müssen lediglich im WAV oder AIFF Format vorliegen. Durch eine der Bibliothek beiliegenden asynchronen Queue können mehrere Sounds nacheinander abgespielt werden. Dieses Feature benutzen wir im Zusammenhang mit den Sprachausgaben bei Erkennung eines Patterns. Da müssen zum Teil zwei Sprachausgaben nacheinander erfolgen, wobei diese den weiteren Verlauf der Steuerungssoftware nicht blockieren dürfen.

Die Software funktioniert allerdings auch selbstständig ohne das Einwirken durch einen Benutzer. Somit ist der Roboter tatsächlich vollautonom.

Eine Kommunikationsschnittstelle über den seriellen Port gibt der Software die Möglichkeit, Befehle beim Erkennen von Patterns über diese Schnittstelle an das Aksenboard zu schicken.

Erkennung der Ausrichtung eines Patterns

Das ARToolkit arbeitet mit einer digitalisierten Matrix des zu erkennenden Patterns. Das bedeutet, dass zu jedem realen Pattern eine digitalisierte Zahlenmatrix existiert, die angibt, welche Pixel dunkel und welche hell sind. Durch die bestimmte Anordnung der dunklen und hellen Pixel, erkennt die Bibliothek durch den Vergleich zwischen digitalisierter Matrix und Abbild selbst ein Pattern.

Beim Erkennen eines Patterns erstellt das ARToolkit ein Zahlenarray, in das die Positionen des Patterns abgespeichert sind. Durch geschicktes Abfragen des Arrays kann man die Ausrichtung berechnen. Das Array beinhaltet Gleitkommazahlen, die nach Fuzzilogik aufgebaut sind. Die Zahlen versuchen einen Zustand nicht als 1 oder 0 darzustellen, sondern erlauben feinere Nuancen zwischen den beiden Werten. Somit zeigt ein Pattern nicht nur genau nach Norden, Süden, Osten und Westen, sondern es sind auch Zwischenzustände möglich.

Im Rahmen dieser Arbeit soll allerdings nur vereinfacht die vier Himmelsrichtungen unterschieden werden. Daher werden die Resultierenden Gleitkommazahlen gerundet. Somit wurde erreicht, dass ein Pattern mit einer nicht ganz waagerechten Ausrichtung dennoch richtig erkannt wird.

Intentionen

Intentionen werden hier als Vorschläge definiert. Dem Roboter erlaubt man, sich selbst um das genaue Ausführen einer Aktion zu kümmern, indem man ihm eine Intention mitgibt. Man braucht also dem Roboter nur sagen: „Fahr nach Links!“ und der Roboter übernimmt eigenverantwortlich das Ansteuern seiner Motoren.

Beim Absetzen von Intentionen ermöglicht man der Steuerungssoftware eine gewisse Ungenauigkeit, die jedoch von der Plattform abgefangen wird. Sollte hierbei ein Hindernis auftauchen, ist es für die Steuerungssoftware, die die Intention mitgeteilt hat, irrelevant. Die Plattform muss das Ausweichen von Hindernissen eigenverantwortlich übernehmen.

Die Befehle, die die Steuerungssoftware an den Mikrocontroller weitergibt, sind als Intentionen gedacht. Somit ist der Entkoppelungsgrad zwischen Steuerungssoftware und Plattform noch ein großes Stück höher und erlaubt zudem einen höheren Grad an Modularität beim Aufbau.

8.3.3. Kommunikation Laptop - Aksenboard

Die Kommunikation zwischen Mikrocontroller und Laptop mit Steuerungssoftware wird, wie bereits erwähnt, über die serielle Schnittstelle realisiert. Die Plattform bringt dafür bereits

vordefinierte Bibliotheksfunktionen mit, während man in Java die Java-COMM-API benutzen kann.

Die Serielle Schnittstelle

Die Fähigkeiten der seriellen Schnittstelle werden durch die Plattform eingeschränkt. Man kann nur mit maximal 9600 Baud kommunizieren. Somit ist die Kommunikation recht langsam, doch immer noch ausreichend für unsere Zwecke.

Wir haben festgestellt, dass die serielle Kommunikation leider nicht so gut funktioniert, da teilweise Steuerungszeichen, die über die Schnittstelle von der Steuerungssoftware an die Plattform geschickt wurden, nicht ankamen. Dies resultiert aus dem Benutzen von Threads auf dem Aksenboard. Somit sahen wir uns gezwungen ein Handshakeverfahren zu implementieren.

Das Handshakeverfahren über die serielle Schnittstelle sieht folgendermaßen aus:

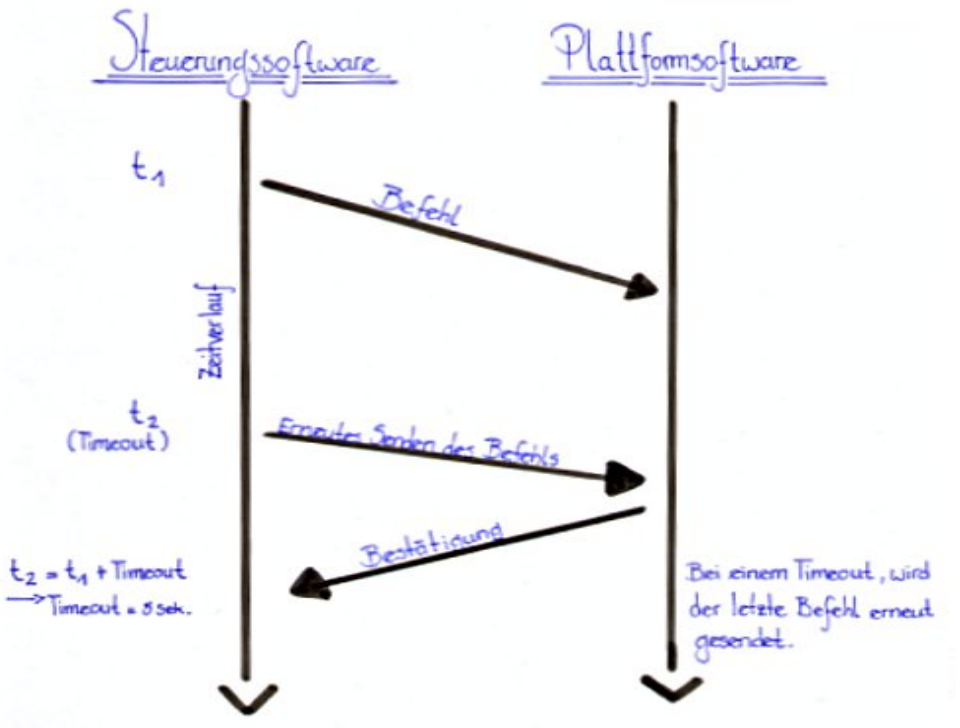


Abbildung 8.13.: Diagramm des Handshake Verfahrens

Jeder Steuerungsbefehl muss von der Plattform durch eine Antwort bestätigt werden. Sollte dies nicht in einer gewissen Zeit passieren (Timeout), wird der Befehl erneut von der Steuerungssoftware an die Plattform geschickt.

Die Java-COMM-API

Die Java-COMM-API ist eine von der Firma SUN bereitgestellte, offizielle Bibliothek. Sie erlaubt es, durch eine Nativ-Schnittstelle direkt auf die serielle Schnittstelle zuzugreifen. Die ankommenden Nachrichten werden durch Java Events sauber implementiert. Dadurch muss nicht auf die Schnittstelle „gepollt“ werden. Die Daten werden in Streams verschickt und empfangen.

Das Einbinden der Bibliothek ist einfach gestaltet und in einer ausführlichen Anleitung, die der Bibliothek beiliegt (sowie etlichen im Internet erhältlichen Tutorials), nachzulesen. Wir verzichten hier bewusst auf eine Erklärung, da sie den Rahmen dieser Arbeit sprengen würde.

Man sollte darauf achten, dass das Empfangen und Verschicken von Daten in separaten Threads implementiert werden sollte.

8.3.4. Benutzen der fertigen Steuerungssoftware

Am Ende unserer Arbeit liegen zwei Programme vor. Das erste Programm ist in C geschrieben und wird über die serielle Schnittstelle auf das Aksenboard gespielt. Das zweite Programm (die Steuerungssoftware) ist in Java geschrieben und läuft auf einem Mini-Laptop, welches ebenfalls auf der Plattform platziert wird. An dem Laptop wird die Kamera angebracht, mit der später die Patterns erkannt werden. Laptop und Aksenboard müssen mit einem seriellen Kabel miteinander verbunden werden. Die Steuerungssoftware wird zuerst initialisiert, dann wird das fertige geflashte Aksenboard angeschaltet und das Programm darauf gestartet.

Wenn alles reibungslos läuft, dann erkennt die Steuerungssoftware, dass das Aksenboard angeschaltet wurde und gibt eine Ausgabe im Logfenster aus.

Das Startsignal wird durch manuelles Drücken der Fernsteuerungstasten in der Oberfläche der Steuerungssoftware oder durch Auslösen eines Ereignisses beim Erkennen eines Patterns ausgelöst. Der Roboter setzt sich daraufhin in Bewegung. Alles weitere geschieht autonom.

Der Roboter fährt in einer ihm unbekannten „feindlichen“ Umgebung und weicht Hindernissen aus. Sobald er ein Pattern erkennt, befolgt er die vorher festgelegte Aktion. Das Erkennen des Patterns zeigt der Roboter durch die Steuerungssoftware im Anzeigebereich als schwebenden dreidimensinalen Kubus, der sich genau auf das Pattern im Bild der Kamera projiziert. Der Roboter bleibt bei der Erkennung stehen und kann, falls so gewünscht und deklariert, eine Sprachdatei abspielen (Audiodatei) und befolgt dann den eigentlichen Befehl, der mit dem Pattern in Verbindung gebracht wurde. Die Befehle und deren Aktionen sind nicht fest einprogrammiert, sondern können frei in einer dem Projekt beiliegenden „Properties“-Datei, unabhängig vom Programmcode, definiert werden. Allerdings kann die Datei nicht mehr zur Laufzeit verändert werden. Die Änderungen hätten keine Auswirkung auf den weiteren Ablauf, da sie einmalig bei Programmbeginn eingelesen werden.

Umfang der Software

Diese Arbeit umfasst zwei voneinander lose abhängige Programme und eine selbst geschriebene Bibliothek. Das eine Programm steuert die Plattform an sich und deren lebenserhaltende Funktionen, das andere steuert das Erkennen von Patterns und die in dem Zusammenhang definierten Aktionen.

Die selbst geschriebene Bibliothek ist eine Funktionensammlung, die es dem Benutzer erlaubt, Sprachdateien (Audio) asynchron über eine Queue abzuspielen.

Alle Teile liegen entweder als fertig kompilierte Programme (C-Programm) oder als gepacktes „JAR“-Archiv vor. Die Plattformsoftware muss nur auf ein Aksenboard übertragen werden. Die Steuerungssoftware wird durch Klicken auf die „start.bat“-Datei gestartet. Darüber hinaus müssen alle Vorbedingungen aus Kapitel 6.2 erfüllt sein.

Definieren von Aktionen

Aktionen werden, wie bereits erwähnt, in einer dem Projekt beiliegender „Properties“-Datei definiert. In der Datei werden auch die Parameter für die serielle Schnittstelle festgelegt, da auf einem anderen Rechner auch andere serielle Ports aktiv sind.

Die „Properties“-Datei beinhaltet eine ausführliche Beschreibung, wie man Aktionen definiert. Eine Aktion ist bezüglich eines Patterns und seiner Lage eindeutig. Sollten zwei Aktionen, die das gleiche Pattern und die gleiche Lage des selbigen definieren, beschrieben worden sein, wird nur die zuletzt definierte Aktion ausgeführt. Genau das Gleiche gilt für sich überlagernde Aktionen. Sollte eine Aktion definiert worden sein, die ein Pattern beschreibt,

dass nur in nördlicher Richtung erkannt werden soll, dann könnte es durch eine andere Aktion überlagert werden, die das gleiche Pattern beschreibt, aber dieses in sämtlichen Lagen einer bestimmten Aktion bindet.

Es gibt keine Grenze was die Anzahl an definierte Aktionen betrifft (höchstens eine physikalische durch den vorhandenen Arbeitsspeicher des Rechners). Neue Patterns können problemlos eingebunden werden, indem man sie in den Unterordner ./Ressourcen/Pattern ablegt und Aktionen mit dem jeweiligen Patternnamen in der „Properties“-Datei definiert.

Einbinden der Software in andere Projekte

Das Einbinden unserer Software in andere Projekte haben wir so einfach wie möglich gehalten. Es liegen zu allen Teilen der Quellcode sowie jeweils eine „Javadoc“ vor. Die Audio-Bibliothek ist von dieser Arbeit vollständig unabhängig und findet sicherlich viele Anwendungsbereiche. Die Plattform-Software, die auf dem Aksenboard läuft, kann durch neue Funktionen ergänzt werden. Threads sollte man nur vorsichtig einsetzen. Ganz ausgereift ist die Benutzung von Threads auf dem Mikrocontroller noch nicht. Die Steuerungssoftware wurde mit einem Model-View-Controller (Beobachter-Pattern) versehen, so dass die Bereiche Oberfläche, Logik und Steuerung so gut es geht, getrennt sind. Dadurch lassen sich einzelne Teile der Steuerungssoftware wiederverwenden.

Die komplette Steuerungssoftware kann auch andere Geräte steuern, die über eine serielle Schnittstelle verfügen. Über Bluetooth kann die serielle Schnittstelle auch kabellos erweitert werden und bietet mehr Einsatzmöglichkeiten. Die USB-Kamera kann ebenfalls durch eine Funkkamera ersetzt werden, wodurch die Möglichkeit entsteht, die Steuerungssoftware komplett kabellos von der Plattform zu benutzen. Dies wäre von Vorteil, wenn die Plattform fliegt und man nicht viel Gewicht, wie im Falle eines Zeppelins, mitnehmen könnte.

Teil V.

Ergebnisse

9. Zusammenfassung und Ergebnisse

9.1. Im Rahmen dieser Bachelorarbeit Erreichtes

Diese Bachelorarbeit beschreibt das Konzept eines autonom fliegenden Zeppelins, der anhand einer Kamera, Patterns erkennen kann. Die Realisierung erfolgt prototypisch auf einer fahrenden, vollautonomen Plattform.

9.1.1. Die fertige Hardware

Ein wesentlicher Teil dieser Arbeit liegt in dem Konzept zur Erstellung und Konstruktion eines Zeppelins. Viele der benötigten Bauteile haben wir geplant und in langer Eigenarbeit selbst hergestellt. Der fertige Zeppelin erwies sich zwar als flugtauglich, doch konnte mit den zur Verfügung stehenden Materialien die Flughöhe nicht auf Dauer gehalten werden. Für die Patternerkennung wäre es allerdings nötig gewesen die Höhe mindestens für 10 Minuten zu halten. Außerdem waren die Kosten für das Helium mit ca. 50 EUR pro Betankung entschieden zu teuer, um aufwendige Tests mit dem Zeppelin und der Patternerkennung durchzuführen.

Die fahrende Plattform, über die wir dann die Patternerkennung realisiert haben, soll für den in dieser Arbeit entstandenen Softwareanteil, ein Ersatz für den Zeppelin darstellen. Die Tests wurden dann durch diese Plattform erschwinglich, wobei die Adaptation auf diese Plattform und der damit einhergehende Aufwand aus Sicht der Steuerungssoftware nur gering ausfielen.

Die Plattform ist autonom, da sie ganz ohne Benutzereingreifen auskommt. Es ist durchaus möglich, die gleiche Steuerungssoftware und sogar die identische Plattformsoftware für den Zeppelin zu benutzen. Wobei nur kleinere Änderungen vorgenommen werden müssten. Man hätte, wie in Kapitel 8.3 beschrieben, lediglich die serielle Schnittstelle über Bluetooth verlängern und die USB-Kamera durch eine Funkkamera ersetzen müssen.

9.1.2. Die fertige Software

Wir haben es geschafft, das doch sehr umfangreiche freie ARToolkit Projekt und dessen Java Schnittstelle JARToolkit so zu benutzen, dass wir die darin enthaltende Funktionalität des Erkennens von Pattern, zum Ansteuern eines fahrenden Roboters benutzt haben. Uns ist es gelungen, die Lage der Pattern so aufzuschlüsseln, dass man Aktionen nicht nur nach den verschiedenen Pattern, sondern auch nach deren Lage definieren kann.

Eine eigene Bibliothek ermöglicht das asynchrone Abspielen von Sprachdateien (WAV, AIFF). Diese Bibliothek kann man ganz unabhängig von diesem Projekt benutzen.

Die Software für die Plattform beinhaltet das Steuern zweier Threads, was wohl auch gleichzeitig die größte Schwierigkeit darstellte. Die saubere Trennung der Threads und das Synchronisieren durch Semaphoren birgt einige Probleme, da das Ausführen von Threads auf dem Aksenboard noch nicht ausgereift ist. In diesem Zusammenhang wurde ein eigenes „serielles Handshake“ implementiert, welches Befehle von der Steuerungssoftware einzeln bestätigt.

Einige ausgefeilte Raffinessen verbergen sich noch zusätzlich in der Software. Die alle hier zu erwähnen wäre zu umfangreich. Einen Blick in den sehr gut kommentierten Quellcode sowie in die „Javadoc“ sollte man nicht auslassen, wenn man die Software benutzen möchte.

Es ist uns auch sehr wichtig gewesen, dass man die Software wiederverwenden kann. Daher haben wir die meisten Einstellungen in einer „Properties“-Datei konfiguriert. Auch die Definition der Aktionen wird in dieser Datei vorgenommen. Somit muss der Quellcode nicht weiter überarbeitet werden, würde man das Projekt anderweitig einsetzen.

Ein weiterer Vorteil unserer Arbeit ist, dass wir die Steuerung mit einer in Java programmierten Software realisiert haben. Dadurch ist sie nicht nur plattformunabhängig, sondern leichter verständlich und für Studenten, die diese Arbeit fortsetzen wollen, ohne größere Einarbeitung begreiflich.

9.2. Soll-Ist-Vergleich

Wir haben uns vorgenommen, einen autonomen Zeppelin mit Patternerkennung zu konstruieren und das Ganze als Bachelorarbeit niederzuschreiben.

Durch Einschränkungen bei der Materialwahl, durch einen geringen Zeitfaktor und durch ein geringes Budget, sind wir gezwungen worden, den Zeppelin lediglich als Konzept darzustellen, während wir den Softwareanteil dieser Arbeit auf eine realisierbare Plattform umsetzen

mussten. An der Software an sich hat sich nicht viel geändert lediglich die Hardware ist ausgetauscht worden.

Insgesamt gesehen haben wir das erreicht, was wir uns in dieser Arbeit vorgenommen haben. An einigen Stellen haben wir sogar die ursprünglich geplante Arbeit ausgedehnt (z.B. die Bibliothek zum Abspielen von Sprachdateien).

Das unabhängige Definieren von Aktionen für verschiedene Pattern, macht die Software vielseitig und schnell anpassungsfähig.

9.3. Erkenntnisse

Der gesamte Entwicklungsprozess, der ein hohes Maß an Konzentration in Anspruch nahm, bescherte uns immer wieder neue Herausforderungen, Chancen und Erkenntnisse. Die wohl größte negative Erkenntnis, die wir erfahren mussten, war die Tatsache, dass unsere Idee nicht allein durch Fleiß zu verwirklichen war, sondern nur unter Einsatz externer finanzieller Mittel realisierbar gewesen wäre.

9.3.1. Finanzielles

Die Finanzierung unseres Zeppelin-Projektes verdanken wir unseren engagierten Professoren, die uns nicht nur betreuten, sondern sich auch bei der Hochschule für Angewandte Wissenschaften erfolgreich für zusätzliche finanzielle Mittel einsetzten. Nur hierdurch war es uns möglich, Zubehör wie eine kabellose Funkkamera und ein Bluetooth-Seriell-Modul zu verwenden.

Trotz dieser dankenswerten Unterstützung sind wir der Meinung, dass es mit Sponsoren aus der Industrie möglich gewesen wäre, hochwertigere Materialien zu verarbeiten. Mit einer professionell angefertigten, passgenauen Ballonhülle wäre z.B. die Zuverlässigkeit des zentralen Bauteils unseres Zeppelins gewährleistet gewesen. Die Realität zeigt, dass sich die zu geringen Mittel, das Einsparen an Materialkosten und damit an der Qualität, entscheidend auf den erfolgreichen Verlauf des gesamten Projekts ausgewirkt haben.

9.3.2. Projektarbeit über mehrere Fachgebiete

Unser Projekt erforderte intellektuelle und handwerkliche Fähigkeiten aus den unterschiedlichsten, teilweise fachfremden Bereichen. Wir haben Bauteile selbst angefertigt, die sonst Maschinenbauer oder Elektrotechniker konstruieren würden. Somit mussten wir von Anfang

an sicherstellen, dass alle Schnittstellen auf einander abgestimmt sind. Ohne eine vorweggenommene Planung der Schnittstellen und Funktionen von einzelnen Komponenten wären beim späteren Zusammenführen erhebliche Probleme aufgetreten.

Als Beispiel kann man die Motorsteuerung anführen. Hier mussten wir schon vor der Konstruktion der Aufhängung kalkulieren, welche Bewegungen später notwendig sein würden. Bevor wir überhaupt mit dem Programmieren anfangen konnten, mussten wir über eine elektronische Schaltung sicherstellen, dass auch der gewünschte Strom die Motoren erreicht.

Die Zusammenarbeit von völlig heterogenen Komponenten war mit Abstand am interessantesten, aber auch am anfälligsten für Fehler.

9.3.3. Software, Frameworks und Entwurfsmuster

Ein wesentlicher Bestandteil unserer Arbeit war neben der eigentlichen Konstruktion, das Implementieren einer Steuerungssoftware, die verschiedene Software Komponenten zusammenbringt.

Da sich das Zusammenführen der verschiedenen Softwarekomponenten als fehleranfällig und sehr zeitaufwendig erwies, haben wir, soweit es möglich war, Entwurfsmuster eingesetzt. Hierdurch mussten wir nicht wiederholt neue Ideen einsetzen, sondern konnten auf bewährte Strukturen zurückgreifen. Speziell das Beobachter-Pattern war enorm hilfreich, weil es uns ermöglichte, die Logik für die Bewegungsabläufe von der Grafischen Oberfläche zu trennen.

Bei der Software mussten wir auf OpenSource und freierhältliche Pakete zurückgreifen. Die von uns verwendeten Programme waren im Wesentlichen Entwicklungsumgebungen für die unterschiedlichsten Programmiersprachen wie Eclipse, Visual Studio und Cygwin. Da wir schon im Laufe des Studiums damit gearbeitet haben, lief hiermit alles reibungslos.

Mit dem ARToolkit mussten wir ein fertiges Framework in unserem System implementieren. Dieses war leider schlecht dokumentiert. So war es zum einen sehr schwierig das Framework überhaupt zu installieren und zum anderen noch schwieriger, Schnittstellen zu unserer Steuerungssoftware zu implementieren. Die Tatsache aber, dass alle unsere Anforderungen an eine Bilderkennungssoftware hier schon vorhanden waren und wahrscheinlich auch alle möglichen schon einmal erarbeitet und beseitigt wurden, überzeugten uns, dieses Framework dennoch zu verwenden. Bei mehr Alternativen zu einem Framework würden wir in Zukunft immer das vorziehen, welches besser dokumentiert ist. Eine kommerzielle Software, die ähnliches leistet wie das ARToolkit, gibt es auch, kostet allerdings mehrere tausend Euro und ist somit unerschwinglich im Hochschulumfeld.

9.3.4. Soziales

Ein so umfassendes, fächerübergreifendes Projekt hätten wir nicht ohne kompetente Hilfe aus den jeweiligen Fachbereichen bewältigen können. So haben wir in dieser Zeit nicht nur fachlich dazu gelernt. Auch die Koordination mit anderen Einrichtungen der HAW wie z.B. dem Institut für Werkstoffkunde, erwies sich als eine spannende Angelegenheit. Man musste fachfremden Leuten einen für Sie verständlichen Überblick über unser Projekt geben, um dann wiederum Ihre Erfahrung effizient in den eigenen Arbeitsprozess einzubinden.

9.3.5. Vorgehen bei Wiederholung des Projektes

Die gesammelten Erkenntnisse dieses Kapitel führen dazu, dass wir das Projekt bei einer möglichen Wiederholung anders angehen würden.

Erster Ansatz wäre hier das Heranziehen von Sponsoren aus der freien Wirtschaft. Unsere Idee ist nach wie vor einmalig. Weder ist sie patentiert noch überhaupt einmal korrekt umgesetzt worden. Somit sollten sich für so eine innovative Idee auch Sponsoren und Interessenten finden lassen, die bereit sind, finanzielle Mittel beizusteuern. Mit aufgestocktem Budget hätten wir dann qualitativ hochwertigere Komponenten kaufen können. Speziell eine fertig verarbeitete Ballonhülle, die unser Budget würde das ganze Projekt aufwerten.

Die einzelnen Arbeiten waren auf sehr viele Fachbereiche verteilt. Die meisten davon gehören nicht zu denen eines Informatikers. Daher wäre es für eine Wiederholung des Projektes unbedingte Vorraussetzung, näher mit den Fachleuten aus den einzelnen Fachbereichen zusammen zuarbeiten. Optimal wäre, wenn einzelne Komponenten komplett von Fachleuten oder zumindest unter ständiger Kontrolle dieser erstellt werden würden. So könnten wir uns als Informatiker auf die Kernaufgaben konzentrieren und Fehlerquellen außerhalb unseres Fachgebiets, würden von Vornherein minimiert werden.

9.4. Probleme

Große Probleme bereiteten uns die Arbeiten auf den diversen fremden Fachgebieten. Zu nennen sind hier speziell die elektrotechnischen und maschinenbautechnischen Arbeiten im Laufe des Projektes. Die Konstruktion von Motoraufhängungen hat uns beispielsweise viel Zeit gekostet, da der Schwierigkeitsgrad der Arbeit für angehende Informatiker sehr hoch ist.

Auch das präzise Messen und Verlöten von Widerständen und „H-Brücken“ gehörte zu den Tätigkeiten, die eine große Herausforderung für uns darstellten. Als Technischer Informatiker haben wir dazu zwar Grundlagenwissen, aber um die richtige H-Brücke für unseren Schaltplan zu bestimmen, haben wir uns sicherheitshalber von Fachleuten Rat eingeholt. Als mögliche Folge hätte durchaus ein Kabelbrand oder Ähnliches auftreten können.

Des Weiteren möchten wir noch einmal die selbst verarbeitete Ballonhülle anführen. Unsere Eigenkonstruktion war zwar gut durchdacht und mit den zur Verfügung stehenden Mitteln auch gut umgesetzt, aber sehr anfällig auf Beschädigungen. Dies hatte unweigerlich zur Folge, dass der Zeppelin Helium verlor und wir davon zu einer neuen Roboterplattform portieren mussten.

Sieht man von diesen Herausforderungen im handwerklichen Bereich ab, hielt die Programmier- und Testphase keine weiteren Schwierigkeiten bereit.

10. Ausblick

10.1. Zukünftige Verwendungsmöglichkeiten

Der im Rahmen dieser Arbeit konzipierte Zeppelin hat eine Vielfalt von Anwendungsmöglichkeiten. Zum einen besticht der Zeppelin durch seine Möglichkeit ohne viel Energie fliegen zu können und zum anderen ist es sehr einfach eine bestimmt Position im dreidimensionalen Raum zu halten. Außerdem ist der Einsatz eines Zeppelins bei großen Menschenmengen, wie z.B. in Messehallen, relativ ungefährlich. Sollte der Zeppelin abstürzen, kann dies nur zu leichten Verletzungen führen, da der Zeppelin erstens nicht viel wiegt und zweitens, keine rotierenden Teile wie bei einem Helikopter vorhanden sind.

Unser Ziel ist es gewesen, einen Zeppelin zu bauen, der in Messehallen autonom umherfliegt, und dabei Werbung für die verschiedenen Stände der Aussteller macht. Dabei sollte sich der Zeppelin anhand von Pattern, die auf den Dächern der Stände und auf dem Boden angebracht sind, orientieren können. Die Werbung hätte dabei auch audiovisueller Natur sein können. Zum einen hätte man an der Außenhülle des Zeppelins Werbeplakate anbringen, zum anderen hätte man Werbung in Form von abspielbaren Audiodateien bereit halten können. Der große Vorteil ist, dass er als vollautonome Plattform keiner ständigen Überwachung bedarf.

Denkbar sind Zeppeline, die bei Massenveranstaltungen zum Einsatz kommen. Dort könnten sie unterschiedlichste Aufgaben übernehmen, wie z.B. Überwachungs- und Suchfunktionen oder Entertainmentzwecke. Der Entwurf unseres Zeppelins eignet sich nur für den Innenbereich. Es ist aber durchaus möglich ein aerodynamisches Design zu implementieren und somit auch den Außenbereich zu bedienen.

Die fahrende Plattform mit der fast gleichen Software wie beim Zeppelin, eignet sich ebenfalls für vielerlei Zwecke. Ein großer Einsatzbereich eines solchen Roboters ist die Lagerverwaltung. Weit verbreitet auf diesem Gebiet ist der Einsatz von Robotern, die sich auf Schienen fortbewegen, da diese einfacher zu programmieren und nicht so fehleranfällig sind.

Durch die Realisierung unseres Projektes zeigen wir, dass es relativ einfach ist, autonom fahrende Roboter mit einer Patternerkennung auszustatten, um somit die Implementierung

von Schienen in einer Lagerhalle zu vermeiden. Dadurch würde man viel Geld und Zeit für die Inbetriebnahme eines solchen Lagersystems sparen.

Die Anzahl an möglichen Einsatzgebieten ist fast grenzenlos. Durch das ARToolkit entstehen weitreichende Möglichkeiten Betrachter, das durch die Kamera Gesehene mit Informationen anzureichern, die gezieltere Entscheidungen zulassen und eine interaktive Arbeitserleichterung darstellen.

10.2. Nachfolgende Projekte

Mit unserer Arbeit haben wir eine komplett autonome Plattform geschaffen, die sich auch leicht um weitere Funktionen erweitern lässt. Die Steuerungssoftware und das JARToolkit sind komplett in Java programmiert und ermöglichen ein flexibles Wechseln des Betriebssytems. Einzige Einschränkung ist das Vorhandensein eines zum Aksenboard passenden C- Compilers.

Nachfolgende Projekte könnten beispielsweise bessere Sensoren verwenden. Bei der Roboter Plattform haben die Sharpsensoren eine kleine Reichweite und die Genauigkeit ist stark abhängig von den Lichtverhältnissen.

Mit den Patterns und der „Properties“-Datei, wo man entsprechende Aktionen hinterlegen kann, sind die Anwendungsmöglichkeiten nahezu unbegrenzt. Der Pool bekannter Patterns kann beliebig erweitert werden. Mit einer verbesserten Spannungsversorgung wären wesentlich aufwendigere und länger andauernde Szenarien denkbar. In Folge dessen wäre auch eine höhere Geschwindigkeit möglich.

Eine optimale Erweiterung wäre eine Funktion, die den Roboter selbst eine Stromquelle finden lässt. So könnte unser finaler Roboter vollkommen unbeaufsichtigt Aktionen ausführen und würde sich bei schwächer werdender Stromversorgung selbstständig wieder aufladen. Dies ließe sich ohne Weiteres über die Patterns realisieren, indem man einem bestimmten Pattern den Ort der Stromquelle hinterlegt.

10.3. Professionelle Möglichkeiten durch Sponsoren aus der Industrie

Wir hatten ein vergleichsweise hohes Budget für eine Bachelorarbeit. Dennoch bleiben immer Sachen offen, die man mit höheren finanziellen Mitteln besser hätte umsetzen können. Wie bereits beschrieben, wären Zuschüsse durch Unternehmen wünschenswert gewesen,

um die Funktionsfähigkeit und Qualität des Projektes in der praktischen Umsetzung zu optimieren.

10.3.1. Material

Beim Material mussten wir keine Kompromisse eingehen, da Balsaholz zu erschwinglichen Preisen zu bekommen war. In der kritischen Auseinandersetzung mit unserer Arbeit diskutierten wir hingegen darüber, ob günstigem Balsaholz nicht zukünftig Carbon vorzuziehen wäre. Das hätte zusätzliches Gewicht gespart und die Stabilität der Konstruktion erhöht.

10.3.2. Verarbeitung

Bei der Verarbeitung hätten wir gerne zumindest die Folie professionell verarbeiten lassen. Wir verfügten über ein Angebot einer fertig geschweißten Ballonhülle in genau unseren Maßen, allerdings zu 4.000 EUR. Mit dieser Folie hätten wir uns einige Probleme gespart. So hätte man den einmal aufgeblasenen Zeppelin auch Tage später noch fliegen lassen können, da kein Helium durch undichte Schweißnähte entwichen wäre. Die Flugkosten wären erheblich gesunken und eine Weiterarbeit am Zeppelin wäre realistisch gewesen. Unsere Kabel waren eindeutig zu dünn und farblich schlecht zu unterscheiden. Für ein aufbauendes Projekt würden wir qualitativ hochwertigere Kabel verwenden, die sich optisch eindeutig unterscheiden lassen.

10.3.3. Komponenten

Das Aksenboard ist sehr performant und eine grundsolide Plattform. Allerdings war es für unsere Zwecke redundant und zu schwer. Es wäre ein speziell auf unsere Bedürfnisse angepasster, extrem leichter Microcontroller optimal gewesen, der sich statt über den seriellen Port, über ein integriertes Bluetooth-Modul flashen lässt und über die Motorports eine höhere Versorgungsspannung liefert.

Unsere Stromversorgung war mit Li-Polymer-Akkus schon sehr hochwertig. Bei einem höheren Budget würden wir aber Akkus mit erhöhter Kapazität empfehlen. Diese kosten erheblich mehr, verlängern dafür aber die Flugzeit deutlich.

Literaturverzeichnis

[Baun 2002] BAUN, Christian: *Latex für Dummies*. Mitp-Verlag, 2002. – ISBN 3-82663-035-1

[Berlin (Zugriff: 06.04.2007)] BERLIN, Technische U.: *MARVIN - Multi-purpose Aerial Robot Vehicle with Intelligent Navigation*. MARVIN - An Autonomously Operating Flying Robot. (Zugriff: 06.04.2007). – URL http://pdv.cs.tu-berlin.de/MARVIN/

[Bock und Knauer 2003] BOCK, Jürgen ; KNAUER, Berthold: *Leichter als Luft - Transport - Trägersysteme. Ballone - Luftschiffe - Plattformen*. Frankenschwelle Verlag, 2003. – ISBN 3-86180-139-6

[Boeing (Zugriff: 29.04.2005)] BOEING: *B17 - Bomber*. B17 - The flying fortress. (Zugriff: 29.04.2005). – URL http://www.b17flyingfortress.de/

[Boersch (Zugriff: 22.08.2005)] BOERSCH, Ingo: *Das Aksen Board*. Labor für künstliche Intelligenz - Fachhochschule Brandenburg. (Zugriff: 22.08.2005). – URL http://ots.fh-brandenburg.de/aksen

[von Braun (Zugriff: 29.04.2005)] BRAUN, Wernher von: *V2 - Vergeltungswaffe 2*. Vergeltungswaffe 2. (Zugriff: 29.04.2005). – URL http://de.wikipedia.org/wiki/A4_(Rakete)

[Drosdowski u. a. 1997] DROSDOWSKI, Prof. Dr. Dr. h.c. Günther (Hrsg.) ; SCHOLZE-STUBENRECHT, Dr. W. (Hrsg.) ; WERMKE, Dr. M. (Hrsg.): *Das Fremdwörterbuch*. Duden, 1997. – ISBN 3-411-04056-4

[Eckel 2003] ECKEL, Bruce: *Thinking in Java*. Markt und Technik, 2003. – ISBN 0-13027-363-5

[Engel und Slapnicar 2002] ENGEL, Stefan ; SLAPNICAR, Klaus W.: *Die Diplomarbeit*. Schäffer-Poeschel Verlag, 2002. – ISBN 3-791-09226-X

[Evening-Standard (Zugriff: 06.04.2007)] EVENING-STANDARD: *Einsatz von Microdrones in der Öffentlichkeit*. Microdrones. (Zugriff: 06.04.2007). – URL http://www.thisislondon.co.uk/news/article-23397459-details/'Flying+saucer'+police+spy+camera+takes+to+the+skies/article.do

[Gamma 2001] GAMMA, Erich: *Entwurfsmuster . Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley, 2001. – ISBN 3-82731-862-9

[Herold 1999] HEROLD, Helmut: *C- Kompaktreferenz*. Addison-Wesley, 1999. – ISBN 3-82731-480-1

[Kleinheins und Meighörner 2004] KLEINHEINS, Peter ; MEIGHÖRNER, Wolfgang: *Die großen Zeppeline. Die Geschichte des Luftschiffbaus: Die Geschichte des Luftschiffbaus*. Springer, Berlin, 2004. – ISBN 3-540-21170-5

[Kolberger und Kolberger (Zugriff: 18.06.2007)] KOLBERGER, Gerd ; KOLBERGER, Tim: *Motorport Modifikation mit H-Brücke*. Labor für künstliche Intelligenz - Fachhochschule Brandenburg. (Zugriff: 18.06.2007). – URL http://ots.fh-brandenburg.de/aksenblog

[Lamb (Zugriff: 22.03.2006)] LAMB, Philip: *ARToolkit Project*. University Washington - Human Interface Technology Lab. (Zugriff: 22.03.2006). – URL http://www.hitl.washington.edu/artoolkit/

[LEO (Zugriff: 12.08.2007)] LEO: *LEO Englisch-Deutsch-Englisch Wörterbuch*. (Zugriff: 12.08.2007). – URL http://dict.leo.org

[Liang 1999] LIANG, Sheng: *The Java Native Interface Programming Guide and Reference*. Addison-Wesley, 1999. – ISBN 0-20132-577-2

[Northrop Grumman's Ryan Aeronautical Center (Zugriff: 29.04.2005)] NORTHROP GRUMMAN'S RYAN AERONAUTICAL CENTER, San D.: *Global Hawk*. U.S. Airforce. (Zugriff: 29.04.2005). – URL http://www.af.mil/factsheets/factsheet.asp?fsID=175

[Oelker (Zugriff: 03.05.2007)] OELKER, Gerhard: *Verstärkerboard für Sharp- Sensoren*. Labor für Angewandte Informatik - Hochschule für Angewandte Wissenschaften. (Zugriff: 03.05.2007). – URL http://users.informatik.haw-hamburg.de/~kvl/sharp/sharp.zip

[Sommerville 2004] SOMMERVILLE, Ian: *Software Engineering*. Addison Wesley / Pearson Education, 2004. – ISBN 3-827-37257-7

[Stollmann 2005] STOLLMANN: *Spezifikation des Bluetooth-Seriell Moduls*. Stollmann (Zugriff: 17.09.2005). 2005. – URL http://www.stollmann.de

[Stuttgart (Zugriff: 07.04.2007)a] STUTTGART, Universität: *LOTTE*. LOTTE Solarluftschiff. (Zugriff: 07.04.2007). – URL http://www.isd.uni-stuttgart.de/lotte/lotte/index.htm

[Stuttgart (Zugriff: 07.04.2007)b] STUTTGART, Universität: *ORCA*. ORCA - Observing Remote Controlled Airship. (Zugriff: 07.04.2007). – URL http://www.isd.uni-stuttgart.de/~haecker/orca/orca_thesis.htm

[SUN (Zugriff: 22.08.2005)] SUN: *Java Comm Package*. Sun Microsystems. (Zugriff: 22.08.2005). – URL http://java.sun.com/products/javacomm/

[Topsicherheit (Zugriff: 13.09.2005)] TOPSICHERHEIT: *Dokumentation der RC Kamera*. Topsicherheit-GmbH. (Zugriff: 13.09.2005). – URL http://www.wes-technik.de

[WES-Technik (Zugriff: 12.09.2005)] WES-TECHNIK: *Dokumentation für Micro Servermotoren*. Systeme und Komponenten für ferngesteuerte Kleinstmodelle. (Zugriff: 12.09.2005). – URL http://www.wes-technik.de

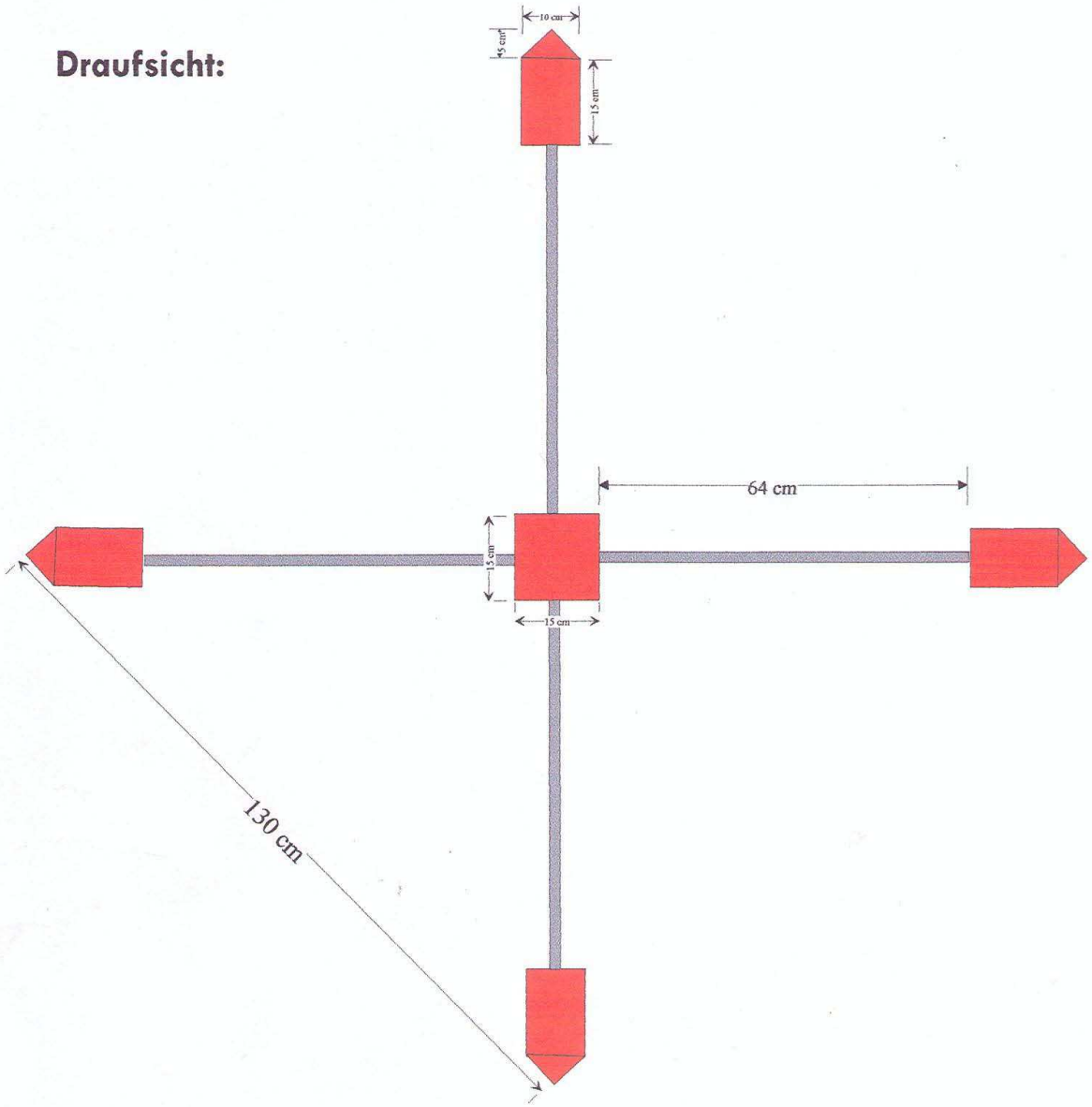
Teil VI.

Anhang

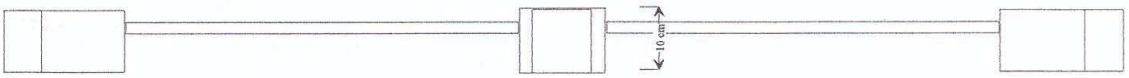
A. Konstruktionspläne des Zeppelins

Zebus Konstruktionsplan

Draufsicht:



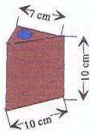
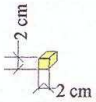
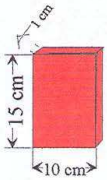
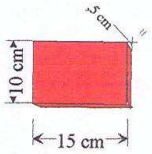
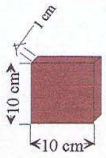
Seitenansicht:



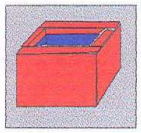
Endmodule:

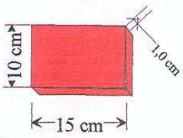
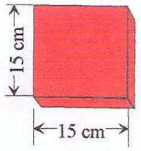
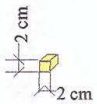
Balseholzdichte : 0,14 g/cm³

Aluminiumdichte : 2,7 g/cm³

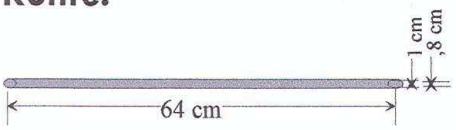
	Volumenberechnungen:	**Masseberechnungen:**
	V = 1/2 * 10 cm * 10 cm * 5 cm - Zillindervol. = 250 cm³ - (PI * r² * 10 cm) = 250 cm³ - (3,14 * 1 cm² * 10 cm) = 250 cm³ - 31,46 cm³ = 218,54 cm³	M = D * V = 0,14 g/cm³ * 218,54 cm³ = 30,59 g
	V = 2 cm * 2 cm * 2 cm = 8 cm³	M = 0,14 g/cm³ * 8 cm³ = 1,12 g
	V = 15 cm * 10 cm * 1 cm = 150 cm³	M = 0,14 g/cm³ * 150 cm³ = 21 g
	V = 15 cm * 10 cm * 0,5 cm = 75 cm³	M = 0,14 g/cm³ * 75 cm³ = 10,5 g
	V = 10 cm * 10 cm * 1 cm = 100 cm³	M = 0,14 g/cm³ * 100 cm³ = 14 g

Mittelmodul:



	Volumenberechnungen:	**Masseberechnungen:**
	V = 15 cm * 10 cm * 1 cm = 150 cm³	M = 0,14 g/cm³ * 150 cm³ = 21 g
	V = 15 cm * 15cm * 1 cm = 225 cm³	M = 0,14 g/cm³ * 225 cm³ = 31,5 g
	V = 2 cm * 2 cm * 2 cm = 8 cm³	M = 0,14 g/cm³ * 8 cm³ = 1,12 g

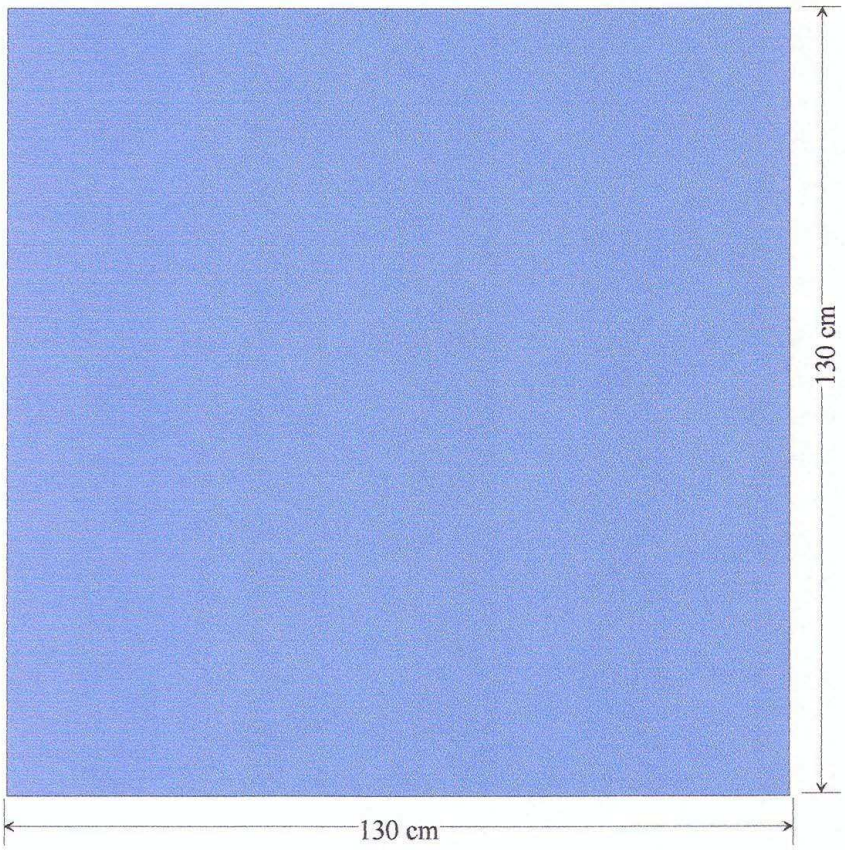
Rohre:



Gewicht (laut Hersteller):

Kohlefaserrohr 10 mm X 8 mm X 1000 mm
= 39 g
Hier: ca. 25 g

Folien Seitenansicht:



Oberfläche = 1,3 m * 1,3 m * 6 Seiten
= 1,69 m² * 6
= 10,14 m²

Gewicht der Folie (laut Hersteller : 30 g/m²

Masse = 30 g/m² * 10,14 m²
= 304,2 g

Berechnungen des Gesamtgewichts :

Gewicht eines Endmoduls: 30,59 g + (1,12 g * 8) + (10,5 * 2) + 21 g + 14 = 95,55 g
Gewicht eines Mittelmoduls : (21 g * 4) + 31,5 g + (1,12 g * 8) = 124,46 g
Gewicht eines Rohres : 25 g
Gewicht der Folie : 304,2 g

Es werden 4 Endmodule und ein Mittelmodul benötigt. Ausserdem werden 4 Rohre verbaut und eine Folie.

Endmodule:	4 *	95,55 g =	382,20 g
Mittelmodul:	1 *	124,46 g =	124,46 g
Rohre:	4 *	25 g =	100,00 g
Elektro Konstrukt:			606,66 g
Folie			304,20 g
Gesamtgewicht			910,86 g

Auftrieb Helium gesamt : 1046 g/m³ * (1,3 m * 1,3 m * 1,3 m)
= 1046 g/m³ * 2,197m³
= 2298,06 g

Nutzlast = Auftrieb gesamt - Gesamtgewicht
= 2298,06 g - 910,86 g
= 1387,20 g

Besorgungsliste Zusammenfassung

Bezeichnung	Anzahl	Kosten/ Stück	Kosten gesamt	Wo
Kok 1100	5	13 €	65 €	http://www.wes-technik.de/Deutsch/Akkus.htm
WES Lader	1	46,50 €	46,50 €	http://www.wes-technik.de/Deutsch/Lader.htm
Pico 5.4	2	22 €	44 €	http://www.wes-technik.de/Deutsch/d-servo.htm
Sharp GP2Y0A02YK	6	24 €	144 €	www.conrad.de
Glacial-tech Silent Blade Lüfter 120 mm	2	12,95 €	25,90 €	www.conrad.de
Kohlefaserrohr 8x10 mm, 1m	4	7 €	28 €	http://www.drachenladen-hennes.de/
Bluetooth Serial Terminal	1	165 €	165€	http://www.wirelessfutures.co.uk/page.php?page=products
Pinhole Cam	1	$ 65	49 €	http://www.spycam4u.com/?source=googleAds
USB 2.0 TV Box	1	37,90 €	37,90 €	http://www.s-datec.com/shop/index.htm
PVC Folie Rollenware, Dicke 0.15 mm	40 * 0,5 (breite) m²	1,38 € / m²	55,20 €	http://www.ballonzauber.de/index1.php
		Total:	660,50 €	

B. Plattformsoftware

Datei: E:\Docs\Study\Bachelor Arbeit\zebus.c 19.08.2007, 10:23:09

```
// Zebus Kontroll Programm für das Aksen Board
// Autor: Bjoern Kaiser und Andre Gratz - Zebus
#include <stdio.h>
#include <regc515c.h>
#include <stub.h>
#include "serielle.h"

unsigned char pid2 = 23,pid3 = 25;
int t1stop = 1; //true: roboter soll auf eingabe warten, danach auto-modus

void robo_vor(void){
    motor_richtung(0,0);
    motor_richtung(1,1);
    motor_pwm(0,10);
    motor_pwm(1,10);
}

void robo_zurueck(void){
    motor_richtung(0,1);
    motor_richtung(1,0);
    motor_pwm(0,10);
    motor_pwm(1,10);
}

//robo dreht sich nach rechts
void robo_rechts(void){
    motor_richtung(0,1);
    motor_richtung(1,1);
    motor_pwm(0,10);
    motor_pwm(1,10);
}

//robo dreht sich nach links
void robo_links(void){
    motor_richtung(0,0);
    motor_richtung(1,0);
    motor_pwm(0,10);
    motor_pwm(1,10);
}

//robo dreht sich nach rechts
void robo_softrechts(){
    motor_richtung(0,0);
    motor_pwm(0,1);
    sleep(700);
    motor_pwm(0,10);
}

//robo dreht sich nach links
void robo_softlinks(){
    motor_richtung(1,1);
    motor_pwm(1,1);
    sleep(700);
    motor_pwm(1,10);
}
//robo hält an - bzw hört auf die motor zu drehen
void robo_stop(void){
    motor_pwm(0,0);
    motor_pwm(1,0);
}
// Hindernis vorne
//vornerechts = a8
//vornemitte = a9
//vornelinks=a11
void fahreOhneAnecken(void){
    int neuwert = 0;
    int sensorwert = -100;
    int groesstersensor = 0;
```

Datei: E:\Docs\Study\Bachelor Arbeit\zebus.c 19.08.2007, 10:23:09

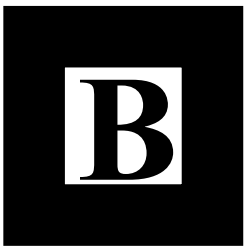
```
    int i =8;
    while(1){
        if(t1stop==1){
            ; //warte auf freigabe durch comcom
            sleep(500);
        }
        else{
            i=8;
            neuwert = 0;
            sensorwert = -100;
            groesstersensor = 0;
            while(i<12){
                neuwert = analog(i);
                if(sensorwert<neuwert){
                    if(i!=10){
                        sensorwert = neuwert;
                        groesstersensor = i;
                    }
                }
                i++;
            }
            //Wenn ein Hinderniss vorhanden
            if(sensorwert>150){
                //Rechter Sensor hat groessten wert
                if(groesstersensor == 8){
                    robo_softlinks();
                }
                //Mittlerer Sensor ist geil
                else if(groesstersensor == 9){
                    robo_stop();
                    sleep(300);
                    robo_links();
                    sleep(300);
                    robo_zurueck();
                    sleep(400);
                    robo_stop();
                }
                //Rechter Sensor ist steil
                else{
                    robo_softrechts();
                }
            }
            robo_vor();
        }//Ende else
  }
}
// Reagiert auf Anweisungen über den seriellen Port
void comCom(void){
    char getPuffer=' ';
    int bestaetigen = 1;
    serielle_puts("Zebus Controller an.");
    while(1){
        bestaetigen = 1;
        getPuffer = serielle_getchar();
        t1stop = 1;
        robo_stop();
        sleep(500);
        lcd_putchar(getPuffer);
        if(getPuffer=='v'){
          robo_vor();
        }
        else if(getPuffer=='z'){
          robo_zurueck();
        }
        else if(getPuffer=='s'){
          robo_stop();
        }
        else if(getPuffer=='l'){
```

Datei: E:\Docs\Study\Bachelor Arbeit\zebus.c 19.08.2007, 10:23:09

```
          robo_links();
        }
        else if(getPuffer=='r'){
          robo_rechts();
        }
        else if (getPuffer=='j'){
          robo_stop();
          bestaetigen = 0;
        }
        else if(getPuffer=='c'){
           robo_stop();
           robo_rechts();
           sleep(700);
        }
        if(bestaetigen){
          sleep(1200);
          robo_stop();
          t1stop = 0; //fahre ohne anecken...
          serielle_putchar('x');
        }
    }
}
//main des Aksen controllers
void AksenMain(void) {
    serielle_init();
    lcd_cls();
    pid2 = process_start(fahreOhneAnecken,50);
    pid3 = process_start(comCom,50);
    process_kill(process_get_pid());
}
```

C. Beispiel Patterns


A

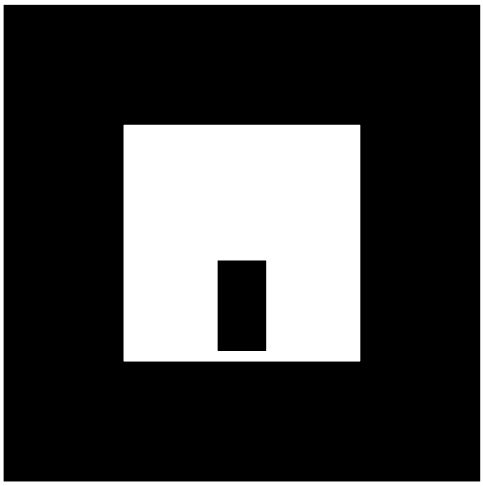

B

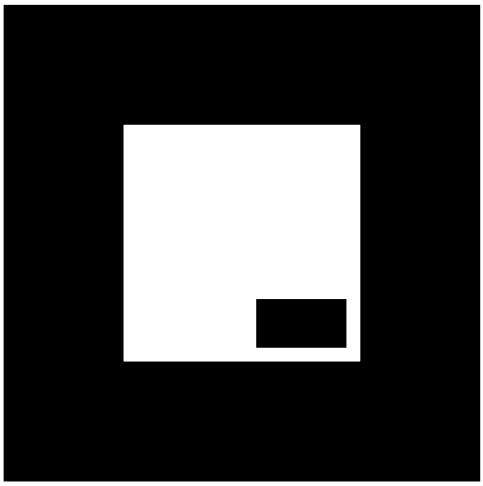

C

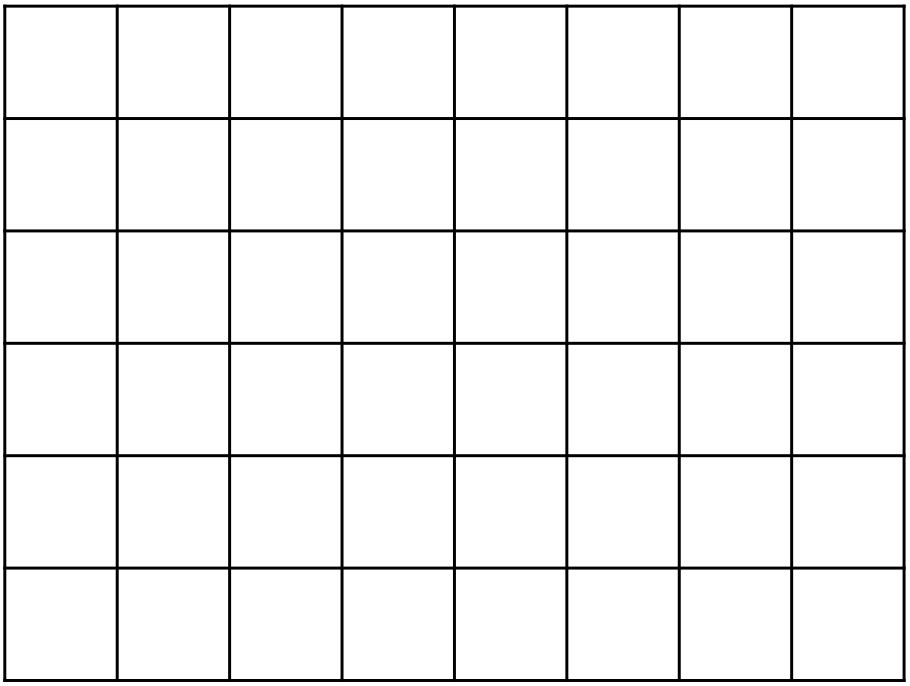

D

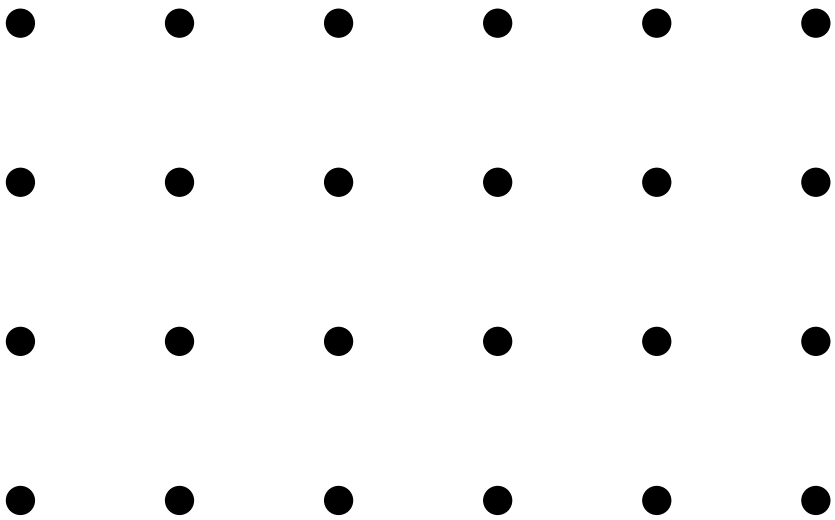

G

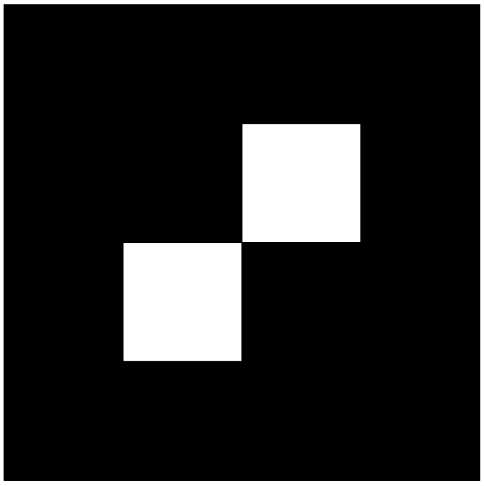

F



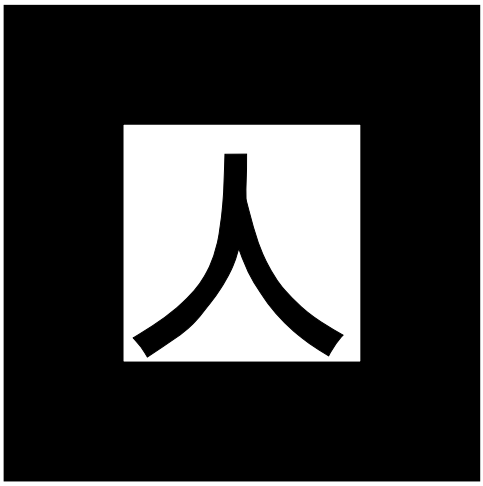









Hiro



D. Steuerungssoftware

```
/**
 * ----------------------------
 * AnimatedCubeFrame.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.Color;
import java.awt.Font;
import java.awt.Graphics;
import java.awt.Graphics2D;
import java.awt.Image;
import java.awt.Toolkit;
import java.net.URL;
import javax.swing.JPanel;

/**
 * @author André Gratz
 *
 */
public class AnimatedCubeFrame extends JPanel implements Runnable,
        MotionObserver {
    private static final long   serialVersionUID    = -8528723066834709476L;
    private static int          CUBE;
    private static final int    SPEED               = 70;
    private int                 cellweite           = 30;
    private int                 celloffset          = 0;
    private int                 bewegungsoffset     = 0;
    private int                 angle               = 0;
    private int                 angle2              = 0;
    private boolean[]           motionstates        =
        { false, false, false, false, false, false};
    private Image               cubeimage           = null;
    Thread                      paint               = new Thread(this);

    /**
     *
     */
    public AnimatedCubeFrame(MotionModel model) {
        super();
        model.addObserver(this);
        try {
            init();
        } catch (Exception e) {
        }
    }

    private void init() {
        this.setBackground(Color.WHITE);
        URL uri = this.getClass().getResource("cubus_logo.gif");
        if (null != uri) {
            cubeimage = Toolkit.getDefaultToolkit().getImage(uri);
        }
        CUBE = this.getHeight() / 2;
        paint.setDaemon(true);
        paint.start();
    }

    public void paint(Graphics g) {
        super.paint(g);
        Graphics2D g2 = (Graphics2D) g;
        int weite = this.getWidth();
        int hoehe = this.getHeight();
        CUBE = hoehe / 3;
        // Gitter
        g2.setColor(Color.BLACK);
```

```
        for (int i = celloffset; i < weite; i += cellweite) {
            g2.drawLine(i + angle2, 0, i + angle, hoehe);
        }
        for (int i = celloffset; i < hoehe; i += cellweite) {
            g2.drawLine(0, i + bewegungsoffset + angle, weite, i + bewegungsoffset
                    + angle2);
        }
        // Cube
        if (null != cubeimage) {
            CUBE = hoehe - 40;
            g2.drawImage(cubeimage, weite / 2 - CUBE / 2, hoehe / 2 - CUBE / 2, CUBE,
                    CUBE, this);
        } else {
            g2.setColor(Color.BLUE);
            g2.fillRect(weite / 2 - CUBE / 2, hoehe / 2 - CUBE / 2, CUBE, CUBE);
            String zebus = "Zebus";
            g.setFont(new Font("Arial", Font.BOLD, 10));
            int fweite = g.getFontMetrics().stringWidth(zebus);
            g2.setColor(Color.WHITE);
            g2.drawString(zebus, weite / 2 - fweite / 2, hoehe / 2 + 4);
        }
    }

    public void run() {
        while (true) {
            if (motionstates[0]) { // Left
                if (angle2 < 300) {
                    angle2 += 10;
                    celloffset -= 6;
                } else {
                    angle2 = 0;
                    celloffset = 0;
                }
            }
            if (motionstates[1]) { // Right
                if (angle < 300) {
                    angle += 10;
                    celloffset -= 6;
                } else {
                    angle = 0;
                    celloffset = 0;
                }
            }
            if (motionstates[2]) { // Forwards
                if (bewegungsoffset < cellweite - 4)
                    bewegungsoffset += 4;
                else
                    bewegungsoffset = 0;
            }
            if (motionstates[3]) { // Backwards
                if (bewegungsoffset > 4)
                    bewegungsoffset -= 4;
                else
                    bewegungsoffset = cellweite;
            }
            if (motionstates[4]) { // UP
                if (cellweite > 30) {
                    cellweite -= 4;
                    if (celloffset > 30)
                        celloffset = 0;
                    else
                        celloffset += 2;
                } else {
                    cellweite = 100;
                    celloffset = 0;
                }
```

```
            }
            if (motionstates[5]) { // Down
                if (cellweite < 100) {
                    cellweite += 2;
                    celloffset -= 6;
                } else {
                    cellweite = 30;
                    celloffset = 0;
                }
            }
            this.repaint();
            try {
                Thread.sleep(SPEED);
            } catch (InterruptedException e) {
            }
        }
    }

    public void setStates(int state) {
        for (int i = 0; i < motionstates.length; i++) {
            if (i == state)
                motionstates[i] = true;
            else
                motionstates[i] = false;
        }
        cellweite = 30;
        celloffset = 0;
        bewegungsoffset = 0;
        angle = 0;
        angle2 = 0;
    }

    public void changeMotion(int motiontype) {
        setStates(motiontype);
        switch (motiontype) {
            case (MotionModel.LEFT):
                if (paint.isInterrupted()) {
                    paint.start();
                }
                break;
            case (MotionModel.RIGHT):
                if (paint.isInterrupted()) {
                    paint.start();
                }
                break;
            case (MotionModel.FORWARDS):
                if (paint.isInterrupted()) {
                    paint.start();
                }
                break;
            case (MotionModel.BACKWARDS):
                if (paint.isInterrupted()) {
                    paint.start();
                }
                break;
            case (MotionModel.STOP):
                if (paint.isAlive()) {
                    paint.interrupt();
                }
                break;
            default:
                ;
        }
    }
}
```

```
/**
 * ----------------------------
 * ARFrame.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.Dimension;
import javax.swing.BorderFactory;
import javax.swing.JPanel;


public class ARFrame extends JPanel {
    private static final long   serialVersionUID    = -7482333851143802970L;
    public ARFrame(){
        super();
        try {
            init();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    public void init(){
        this.setPreferredSize(new Dimension(240, 150));
        this.setLayout(null);
        this.setBorder(BorderFactory.createTitledBorder("Kameraübertragung"));
    }
}
```

```
/**
 * ----------------------------
 * ARViewerPanel.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.BorderLayout;
import java.awt.GraphicsConfiguration;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.io.File;

import javax.media.j3d.AmbientLight;
import javax.media.j3d.Appearance;
import javax.media.j3d.BoundingSphere;
import javax.media.j3d.Bounds;
import javax.media.j3d.BranchGroup;
import javax.media.j3d.Canvas3D;
import javax.media.j3d.DirectionalLight;
import javax.media.j3d.Locale;
import javax.media.j3d.TransformGroup;
import javax.media.j3d.VirtualUniverse;
import javax.swing.JPanel;
import javax.swing.JPopupMenu;
import javax.vecmath.Color3f;
import javax.vecmath.Point3d;
import javax.vecmath.Vector3f;

import net.sourceforge.jartoolkit.java3d.Behavior.ARBehavior;
import net.sourceforge.jartoolkit.java3d.util.JARToolKit3D;

import com.sun.j3d.utils.geometry.ColorCube;
import com.sun.j3d.utils.universe.SimpleUniverse;
import com.zebus.jartoolkit.data.pattern.PattAktion;

public class ARViewerPanel extends JPanel implements KeyListener {
    private static final long   serialVersionUID    = -3421061411259941653L;
    private Bounds              bounds              = new BoundingSphere(new Point3d(
        0.0, 0.0, 0.0),
        Double.MAX_VALUE);

    // Drawing canvas for 3D rendering
    public Canvas3D             canvas;
    // BranchGroup for the Content Branch of the scene
    private BranchGroup         contentBranch;
    // TransformGroup node of the scene contents
    private TransformGroup      contentsTransGr;
    // Locale of the scene graph.
    private Locale              locale;
    private ARBehavior          m_arBehavior        = null;
    private JARToolKit3D        m_JARToolKit3D      = null;
    private TransformGroup[]    objTrans;
    // Virtual Universe object.
    private VirtualUniverse     universe;

    public PattAktion[]         pattern_array       = {};

    // The instance of ZCP
    public ARViewerPanel(ZCP zcp){

        // Creating and setting the Canvas3D
        GraphicsConfiguration config = SimpleUniverse.getPreferredConfiguration();
        canvas = new Canvas3D(config);
        this.setLayout(new BorderLayout());
        this.add(canvas, "Center");
```

```
        // Setting the VirtualUniverse and the Locale nodes
        setUniverse();
        try {
            m_JARToolKit3D = JARToolKit3D.create();
            // The instance of ZCP
            m_JARToolKit3D.setZCP(zcp);
            m_JARToolKit3D.initialize(new File(System.getProperty("user.dir")
                    + "/resources/pattern/camera_para.dat").getAbsolutePath());
            // Create the viewing branch
            locale.addBranchGraph(m_JARToolKit3D.createViewingBranch(canvas));
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(0);
        }
        // Setting the Pattern Array
        this.pattern_array = zcp.getPatternAktion();
        // Setting the content branch
        setContent();
        // To avoid problems between Java3D and Swing
        JPopupMenu.setDefaultLightWeightPopupEnabled(false);
        canvas.addKeyListener(this);
        addKeyListener(this);
        setSize(m_JARToolKit3D.getWidth(), m_JARToolKit3D.getHeight());
    }

    public void keyPressed(KeyEvent e) {

    }

    // EventCatcher for keyboard input
    // up and down change the treshold
    public void keyReleased(KeyEvent e) {
        switch (e.getKeyCode()) {
        case KeyEvent.VK_UP:
            m_arBehavior.setThreshold(m_arBehavior.getThreshold() + 10);
            // System.out.println("threshold = " + m_arBehavior.getThreshold());
            break;
        case KeyEvent.VK_DOWN:
            m_arBehavior.setThreshold(m_arBehavior.getThreshold() - 10);
            // System.out.println("threshold = " + m_arBehavior.getThreshold());
            break;
        default:
            return;
        }
    }

    public void keyTyped(KeyEvent e) {
    }

    private void setContent() {
        // Creating the content branch
        contentsTransGr = new TransformGroup();
        contentsTransGr.setCapability(TransformGroup.ALLOW_TRANSFORM_WRITE);
        setLighting();
        try {
            contentsTransGr.addChild(m_JARToolKit3D.createBackground());
            m_arBehavior = m_JARToolKit3D.createRecognition();
            contentsTransGr.addChild(m_arBehavior);
            // Setting for all Patterns to be recognize a colorcube
            Appearance appearance = new Appearance();
            ColorCube[] cube = new ColorCube[pattern_array.length];
            objTrans = new TransformGroup[pattern_array.length];
            for (int i = 0; i < pattern_array.length; i++) {
                objTrans[i] = m_JARToolKit3D.createPatternTransform(pattern_array[i]
                        .getPattPath(), false);
                m_arBehavior.registerObject(objTrans[i]);
```

```
                contentsTransGr.addChild(objTrans[i]);
                cube[i] = new ColorCube(20.0f);
                cube[i].setAppearance(appearance);
                objTrans[i].addChild(cube[i]);

            }

        } catch (Exception e) {
            System.err.println(e);
            System.exit(0);
        }

        contentBranch = new BranchGroup();
        contentBranch.addChild(contentsTransGr);
        // Compiling the branch graph before making it live
        contentBranch.compile();
        // Adding a branch graph into a locale makes its nodes live (drawable)
        locale.addBranchGraph(contentBranch);
    }

    private void setLighting() {
        AmbientLight ambientLight = new AmbientLight();
        ambientLight.setEnable(true);
        ambientLight.setColor(new Color3f(0.10f, 0.1f, 1.0f));
        ambientLight.setCapability(AmbientLight.ALLOW_STATE_READ);
        ambientLight.setCapability(AmbientLight.ALLOW_STATE_WRITE);
        ambientLight.setInfluencingBounds(bounds);
        contentsTransGr.addChild(ambientLight);
        DirectionalLight dirLight = new DirectionalLight();
        dirLight.setEnable(true);
        dirLight.setColor(new Color3f(1.0f, 0.0f, 0.0f));
        dirLight.setDirection(new Vector3f(1.0f, -0.5f, -0.5f));
        dirLight.setCapability(AmbientLight.ALLOW_STATE_WRITE);
        dirLight.setInfluencingBounds(bounds);
        contentsTransGr.addChild(dirLight);
    }

    private void setUniverse() {
        // Creating the VirtualUniverse and the Locale nodes
        universe = new VirtualUniverse();
        locale = new Locale(universe);
    }
}
```

```java
/**
 * ----------------------------
 * ARViewerTester.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.Dimension;

/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 */
public class ARViewerTester extends JFrame{
    /**
     *
     */
    private static final long   serialVersionUID    = -9211942007006130921L;
    private static JFrame       frame;

    /**
     *
     */
    public ARViewerTester() {
        super();
    }

    public static void createAndShowGUI() {
        frame = new JFrame("ARViewerTester");
        frame.setSize(new Dimension(401, 370));

        ARViewerPanel contentPane = new ARViewerPanel(new ZCP());
        try {
            URL uri= ARViewerTester.class.getResource("data/images/cubus_logo.gif");
            Image logo = Toolkit.getDefaultToolkit().getImage(uri);
            frame.setIconImage(logo);
        } catch (NullPointerException e1) {
        }
        contentPane.setOpaque(true);
        frame.setContentPane(contentPane);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                System.exit(0);
            }
        });
    }

    public static void main(String[] args) {
        JFrame.setDefaultLookAndFeelDecorated(true);
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                //createAndShowGUI();
            }
        });
    }
}
```

```
/**
 * ----------------------------
 * ComInterface.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.io.IOException;

public class ComInterface implements SerialPortEventListener {
    public static String        PORT1   = "COM";
    private CommPortIdentifier  portId;
    private Enumeration         portList;
    private InputStream         inputStream;
    private OutputStream        outputStream;
    private SerialPort          serialPort;
    private ZCP                 zcp;
    private int                 speed   = 9600;
    private int                 bits    = SerialPort.DATABITS_8;
    private int                 sbit    = SerialPort.STOPBITS_1;
    private int                 parity  = SerialPort.PARITY_NONE;
    private int                 portnr  = 2;

    public ComInterface(ZCP zcp) {
        this.zcp = zcp;
        int[] props_array = zcp.getComProps();
        this.portnr = props_array[0];
        this.speed = props_array[1];
        this.bits = props_array[2];
        this.sbit = props_array[3];
        this.parity = props_array[4];
        this.portList = CommPortIdentifier.getPortIdentifiers();
        while (portList.hasMoreElements()) {
            portId = (CommPortIdentifier) portList.nextElement();
            if (portId.getPortType() == CommPortIdentifier.PORT_SERIAL) {
                if (portId.getName().equals(PORT1 + portnr)) {
                    try {
                        serialPort = (SerialPort) portId.open("ARViewer", 2000);
                        serialPort.addEventListener(this);
                        serialPort.notifyOnDataAvailable(true);
                        serialPort.setSerialPortParams(speed, bits, sbit, parity);
                        inputStream = serialPort.getInputStream();
                        outputStream = serialPort.getOutputStream();
                    } catch (PortInUseException pe) {
                        pe.printStackTrace();
                    } catch (TooManyListenersException te) {
                        te.printStackTrace();
                    } catch (IOException io) {
                        io.printStackTrace();
                    } catch (UnsupportedCommOperationException ue) {
                        ue.printStackTrace();
                    }
                }
            }
        }
    }

    public void serialEvent(SerialPortEvent event) {

        switch (event.getEventType()) {
        case SerialPortEvent.BI:
        case SerialPortEvent.OE:
        case SerialPortEvent.FE:
        case SerialPortEvent.PE:
        case SerialPortEvent.CD:
        case SerialPortEvent.CTS:
```

```
            case SerialPortEvent.DSR:
            case SerialPortEvent.RI:
            case SerialPortEvent.OUTPUT_BUFFER_EMPTY:
            case SerialPortEvent.DATA_AVAILABLE:
                try {
                    int buffersize = inputStream.available();
                    byte[] readbuffer = new byte[buffersize];
                    if (buffersize > 0) {
                        inputStream.read(readbuffer);
                        String buffer = new String(readbuffer);
                        zcp.writeToLog("Zebus sendet: " + buffer);
                        if (buffer.equals("x")) {
                            ZCP.aksenJobDone = true;
                        }
                        inputStream.reset();
                    }
                } catch (IOException e) {
                    // e.printStackTrace();
                }
                break;
            }
        }

        public void out(String toWrite) throws IOException {
            outputStream.write(toWrite.getBytes());
        }

        public void closeStreams() throws IOException {
            inputStream.close();
            outputStream.close();
        }
    }
```

```java
/**
 * ----------------------------
 * MotionModel.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.util.ArrayList;
import java.util.Iterator;

/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 *          Implemented Observer Pattern to treat direction changes
 *
 */
public class MotionModel {
    /** Alias for non action */
    public static final int EMPTY       = -1;
    /** Alias for direction "forward" */
    public static final int FORWARDS    = 1;
    /** Alias for direction "right" */
    public static final int RIGHT       = 2;
    /** Alias for direction "backward" */
    public static final int BACKWARDS   = 3;
    /** Alias for direction "left" */
    public static final int LEFT        = 4;
    /** Alias for driving in circle */
    public static final int CIRCLE      = 5;
    /** Alias for direction fullstop */
    public static final int STOP        = 6;
    /** Alias for direction fullstop without response */
    public static final int STOP_WR     = 7;

    /**
     * Integer, that saves the current moving direction. Set to private with
     * access only through setter and getter methods.
     */
    private int                         current_motiontype;

    /**
     * ArrayListe with signed in oberservers. All observers saved here, get
     * noticed immediately after changing the moving direction
     */
    private ArrayList               observers   = new ArrayList();

    /** public default construktor */
    public MotionModel() {
        super();
    }

    /**
     * @param o
     *          Object, to be signed in as new observer.
     */
    @SuppressWarnings("unchecked")
    public void addObserver(Object o) {
        observers.add(o);
    }

    /**
     * @param o
     *          Object, to be removed from the observer list
     */
    public void removeObserver(Object o) {
        observers.remove(o);
```

```
    }

    /**
     * @param motiontype
     *          Alias for the moving direction to be set. All signed in observers
     *          get a notice that the moving direction changed.
     * @see com.zebus.jartoolkit.MotionModel#current_motiontype
     * @see com.zebus.jartoolkit.MotionModel#observers
     */
    public void setNewMotion(int motiontype) {
        try {
            MotionObserver obs = null;
            Iterator it = observers.iterator();
            // updating all observers
            while (it.hasNext()) {
                obs = (MotionObserver) it.next();
                obs.changeMotion(motiontype);
            }
            this.current_motiontype = motiontype;
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * @return Alias containing the current moving direction. Getter method for
     *         current_motiontype.
     * @see com.zebus.jartoolkit.MotionModel#current_motiontype
     */
    public int getCurrent_motiontype() {
        return current_motiontype;
    }

    /**
     * @param Alias
     *          containing the current moving direction. Setter method for
     *          current_motiontype .
     * @see com.zebus.jartoolkit.MotionModel#current_motiontype.
     */
    public void setCurrent_motiontype(int motiontype) {
        this.current_motiontype = motiontype;
    }
}
```

```
/**
 * ----------------------------
 * MotionObserver.java
 * ----------------------------
 */
package com.zebus.jartoolkit;
/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 * Interface with the method "changeMotion". Computes the movements.
 */
interface MotionObserver {

    /**
     * @param motiontype Integer, to set the moving direction.
     */
    public void changeMotion(int motiontype);
}
```

```
/**
 * ----------------------------
 * NavigationPanel.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.Dimension;

/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 */

public class NavigationPanel extends JPanel implements ActionListener,
        KeyListener, MotionObserver {

    private static final long   serialVersionUID    = -7482333851143802970L;
    private JButton[]           button              = new JButton[7];
    private MotionModel         model;

    public NavigationPanel(MotionModel model) {
        super();
        this.model = model;
        model.addObserver(this);
        try {
            init();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private void init() {
        this.setPreferredSize(new Dimension(170, 130));
        this.setLayout(null);
        this.setBorder(BorderFactory.createTitledBorder("Navigation"));
        init_buttons();
    }

    private void init_buttons() {
        String[] labels = { "<", ">", "^", "V", "H", "R", "+" };
        int[] hoffset = { 10, 70, 40, 40, 130, 130, 40 };
        int[] voffset = { 60, 60, 30, 90, 30, 90, 60 };
        int bweite = 30;
        int bhoehe = 30;
        for (int i = 0; i < button.length; i++) {
            button[i] = new JButton();
            button[i].setFocusPainted(false);
            button[i].setMargin(new Insets(0, 0, 0, 0));
            button[i].addActionListener(this);
            button[i].addKeyListener(this);
            button[i].setText(labels[i]);
            button[i].setBounds(hoffset[i], voffset[i], bweite, bhoehe);
            this.add(button[i]);
        }
    }

    public void actionPerformed(ActionEvent e) {
        for (int i = 0; i < button.length; i++) {
            if (e.getSource().equals(button[i])) {
                perform(i);
            }
        }
    }

    public void keyTyped(KeyEvent arg0) {
```

```
        // Do nothing here
    }

    public void keyPressed(KeyEvent arg0) {
        int option = arg0.getKeyCode();
        int[] key = { KeyEvent.VK_LEFT, KeyEvent.VK_RIGHT, KeyEvent.VK_UP,
                KeyEvent.VK_DOWN, KeyEvent.VK_PAGE_UP, KeyEvent.VK_PAGE_DOWN,
                KeyEvent.VK_ESCAPE };
        for (int i = 0; i < button.length; i++) {
            if (option == key[i])
                perform(i);
        }
    }

    public void keyReleased(KeyEvent arg0) {
        // Do nothing here
    }

    private void perform(int option) {
        switch (option) {
            case (0): //left
                model.setNewMotion(MotionModel.LEFT);
                break;
            case (1): //right
                model.setNewMotion(MotionModel.RIGHT);
                break;
            case (2): //forwards
                model.setNewMotion(MotionModel.FORWARDS);
                break;
            case (3): //backwards
                model.setNewMotion(MotionModel.BACKWARDS);
                break;
            case (6): //stop
                model.setNewMotion(MotionModel.STOP);
                break;
            default:
                ;
        }
    }

    public void changeMotion(int motiontype) {

    }
}
```

```java
/**
 * ----------------------------
 * PropertyManager.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.io.File;

/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 */
public class PropertyManager {
    private static final String PROPS           = "resources/arviewer.properties";
    public static final String  PAT_TEMPLATE    = "pattern";
    private PattAktion[]        pattaktion;
    private int[]               com_props;

    /**
     * Konstruktor.
     */
    public PropertyManager() {
        super();
        Properties loadedProps = loadProperties();
        if (null != loadedProps) {
            mapProperties(loadedProps);
        }
    }

    /**
     * Loads all properties of the properties file. The properties file should be
     * placed in the current application directory.
     *
     * @return all properties found in the properties file.
     */
    public Properties loadProperties() {
        Properties tprops = new Properties();
        File propsfile = new File(System.getProperty("user.dir"), PROPS);
        try {
            FileInputStream fin = new FileInputStream(propsfile);
            tprops.load(fin);
            fin.close();
        } catch (FileNotFoundException e) {
            return null;
        } catch (IOException e) {
            return null;
        }
        return tprops;
    }

    /**
     * Maps the properties.
     *
     * @param props
     *          The properties file to be analysed.
     */
    public void mapProperties(Properties props) {
        // Com Port wird gemapped
        int com_port = Integer.parseInt(props.getProperty("com_port"));
        int com_speed = Integer.parseInt(props.getProperty("com_speed"));
        int com_bits = Integer.parseInt(props.getProperty("com_bits"));
        int com_sbit = Integer.parseInt(props.getProperty("com_stopbit"));
        int com_parity = Integer.parseInt(props.getProperty("com_parity"));
        int[] tmpcom = { com_port, com_speed, com_bits, com_sbit, com_parity };
        this.com_props = tmpcom;
```

```
        // Patterns werden gemapped
        int counter = 1;
        while (props.containsKey(PAT_TEMPLATE + counter)) {
            counter++;
        }
        this.pattaktion = new PattAktion[--counter];
        for (int i = 0; i < counter; i++) {
            String tmpprop = props.getProperty(PAT_TEMPLATE + (i+1));
            if (null != tmpprop) {
                String[] array = convertCSVToArray(tmpprop, ";");
                String[] audio_array = {array[3]};
                if(isTextAList(array[3])){
                    audio_array = convertCSVToArray(array[3], ",");
                }
                int dir = Integer.parseInt(array[1]);
                int action = Integer.parseInt(array[2]);
                this.pattaktion[i] = new PattAktion(array[0], audio_array, dir,
action);
            }
        }
    }

    /**
     * Getter method for the Properties Pattern Aktions.
     *
     * @return an Array of Pattern Aktions.
     */
    public PattAktion[] getPattAktion() {
        return this.pattaktion;
    }

    /**
     * @return - com port properties.
     */
    public int[] getComProperties() {
        return this.com_props;
    }

    /**
     * Converter method for a csv list to an array.
     *
     * @param list
     *          A csv list.
     * @return An array containing all elements of the list.
     */
    public String[] convertCSVToArray(String list, String separator) {
        StringTokenizer st = new StringTokenizer(list, separator);
        String[] tmparray = new String[st.countTokens()];
        int counter = 0;
        while (st.hasMoreTokens()) {
            tmparray[counter++] = st.nextToken();
        }
        return tmparray;
    }

    /**
     * Checks if the text is a csv list or not.
     *
     * @param text
     *          A text as String
     * @return true, if the text is a csv list or false if not.
     */
    public boolean isTextAList(String text) {
        if(null == text) return false;
        if (text.indexOf(",") != -1) {
```

```
            return true;
        }
        return false;
    }
}
```

```
/**
 * ----------------------------
 * ZCP.java
 * ----------------------------
 */
package com.zebus.jartoolkit;

import java.awt.BorderLayout;
import java.awt.Dimension;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.io.File;
import java.io.IOException;
import java.net.URL;
import java.util.Date;

import javax.swing.BorderFactory;
import javax.swing.ImageIcon;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;
import javax.swing.SwingConstants;

import com.zebus.jartoolkit.data.pattern.PattAktion;
import com.zebus.voiceplayer.VoicePlayerBean;

/**
 * @author André Gratz & Björn Kaiser
 * @version 1.0<br>
 *          Graphical User Interface to show the whole Control Panel. Inherits
 *          from JPanel (Swing) and implements the MotionOberserver Interface.
 *          The interface is necessary, in order to detect changes in Motion and
 *          react.
 */
public class ZCP extends JPanel implements MotionObserver {

    private static final long   serialVersionUID    = 76057183730133508L;
    /** Container to hold the panel */
    private static JFrame   frame;
    /** TextArea for logmessage output */
    private JTextArea       log;
    /** Container to hold the TextArea in order provide scrolling. */
    private JScrollPane     scroller;
    private JPanel          header, anime, logging;
    /** Headline of the Control Panel */
    private JLabel          headerlabel;
    private MotionModel     model;
    /** Link to the proper logos, shown in the GUI */
    private static URL      uro, uri;
    /** Reference to Voiceplayer. Provides Playing of audiofiles */
    private static VoicePlayerBean  play        = new VoicePlayerBean();
    private ARViewerPanel           arv;
    private static ComInterface     cominterface;
    boolean                         initialized = false;
    private PattAktion[]            pattaktion;
    private int[]                   comprops;

    /**
     * boolean array. Every place holds a boolean and shows wether the pattern is
     * found. <br>
     * e.g.: [2]true: pattern#3 found
     */
```

```
boolean[]               pattern_boolean     = { true, true, true, true, true};
public static boolean   aksenJobDone        = true;

/**
 * Public constructor without argument. Components get initialised here and
 * the Layout is being constructed
 */
public ZCP() {
    PropertyManager pm  = new PropertyManager();
    this.pattaktion     = pm.getPattAktion();
    this.comprops       = pm.getComProperties();
    log                 = new JTextArea();
    scroller            = new JScrollPane(log);
    model               = new MotionModel();
    uro                 = this.getClass().getResource("cubus_logo.gif");
    header              = new JPanel();
    anime               = new JPanel();
    logging             = new JPanel();
    headerlabel         = null;
    uri                 = this.getClass().getResource("logo_haw.gif");
    Image logo          = null;
    ImageIcon icon      = null;
    // seriell writing
    try {
        cominterface = new ComInterface(this);
    } catch (Exception e) {
        writeToLog("Error in the cominterface: " + e.toString());
    }
    if (null != uri) {
        logo = Toolkit.getDefaultToolkit().getImage(uri);
        icon = new ImageIcon(logo);
    }
    if (null != icon) {
        headerlabel = new JLabel(
                "<html><h3><i>Zebus Control Program</i></h3></html>", icon,
                SwingConstants.LEFT);
    } else
        headerlabel = new JLabel();

    this.setLayout(new BorderLayout());
    model.addObserver(this);
    headerlabel.setIconTextGap(30);
    header.add(headerlabel);
    anime.setLayout(new BorderLayout());
    anime.setBorder(BorderFactory.createTitledBorder("Zebus Status"));
    anime.add(BorderLayout.CENTER, new AnimatedCubeFrame(model));
    anime.setPreferredSize(new Dimension(230, 250));

    play.setFileBase(System.getProperty("user.dir") + File.separator
            + "resources" + File.separator + "audios" + File.separator);

    arv = new ARViewerPanel(this);
    arv.setPreferredSize(new Dimension(170, 130));
    arv.setOpaque(true);
    arv.setVisible(true);

    scroller.setPreferredSize(new Dimension(401, 140));
    scroller.setBorder(BorderFactory.createTitledBorder("Logging"));
    logging.setLayout(new BorderLayout());
    logging.add(BorderLayout.CENTER, scroller);
    // Throw the components on the panel
    this.add(BorderLayout.NORTH, header);
    this.add(BorderLayout.EAST, new NavigationPanel(model));
    this.add(BorderLayout.WEST, anime);
    this.add(BorderLayout.CENTER, arv);
    this.add(BorderLayout.SOUTH, scroller);
```

```
        // Throw the status messages on the logpanel
        writeToLog("Log session started.");
    }

    /**
     * @return - git das Pattern Aktion Array zurueck.
     */
    public PattAktion[] getPatternAktion() {
        return this.pattaktion;
    }

    /**
     * @return - Gibt die Com-Port-Properties als int Array zurueck.
     */
    public int[] getComProps() {
        return this.comprops;
    }

    /**
     * @param message
     *          String to be shown on the log panel
     */
    public void writeToLog(String message) {
        Date date = new Date();
        String tanga = log.getText();
        log.setText(date + "   |   " + message + "\n");
        log.append(tanga);
    }

    /**
     * @see com.zebus.jartoolkit.MotionObserver#changeMotion(int)
     */
    public void changeMotion(int motiontype) {
        try {
            switch (motiontype) {
            case (MotionModel.LEFT):
                cominterface.out("l");
                break;
            case (MotionModel.RIGHT):
                cominterface.out("r");
                break;
            case (MotionModel.FORWARDS):
                cominterface.out("v");
                break;
            case (MotionModel.BACKWARDS):
                cominterface.out("z");
                break;
            case (MotionModel.STOP):
                cominterface.out("s");
                break;
            case (MotionModel.STOP_WR):
                cominterface.out("j");
                break;
            case (MotionModel.CIRCLE):
                cominterface.out("c");
                break;
            default:
                ;
            }
        } catch (Exception e) {
            writeToLog("Error in the cominterface: " + e.toString());
        }
    }

    /** Alias for "forward" */
    public static final int NORTH       = 1;
```

```
/** Alias for "right" */
public static final int EAST        = 2;
/** Alias for "backward" */
public static final int SOUTH       = 3;
/** Alias for "left" */
public static final int WEST        = 4;
/** Alias for error */
public static final int FEHLER  = -1;

/**
 * @param x
 *          measured value from the horizontal axis. Derives from matrix array
 *          of ARPatternTransformGroup
 * @param y
 *          measured value from the vertical axis. Derives from matrix array
 *          of ARPatternTransformGroup
 * @return Fixed Alignment is going to be computed from measured values. This
 *         can be compared with computing digital, fixed values from analog,
 *         measured values. The values, measured from the cam are kind of
 *         analog, while the Alignment is kind of a digital value
 */
public int erkannteAusrichtung(double x, double y) {
    long vertikal = Math.round(y);
    long horizontal = Math.round(x);

    if (vertikal == -1 && horizontal == 0) {
        return ZCP.NORTH;
    } else if (vertikal == 1 && horizontal == 0) {
        return ZCP.SOUTH;
    } else if (vertikal == 0 && horizontal == -1) {
        return ZCP.EAST;
    } else if (vertikal == 0 && horizontal == 1) {
        return ZCP.WEST;
    }
    return ZCP.FEHLER;
}

/**
 * @param fileToPlay
 *          static string containing the filename of the file to play
 */
private void playSound(String fileToPlay) {
    play.setAudioName(fileToPlay);
    if (!play.isPlaying()) {
        play.start();
    }
}

/**
 * Sleep function. Provides some rest to the process. Nice during some polling
 * loops
 *
 * @param sec
 *          long value containing the seconds to sleep.
 */
private void waitTime(long sec) {
    try {
        Thread.sleep(sec);
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
}

/**
 * Control Method, that handle the "patternFound" Event. Here the decision is
 * made, how to react on the found pattern. Has to be synchronized, because
```

```
     * there may be several threads trying to access the method. But only one is
     * permitted
     *
     * @param pattern
     *          Static String representation of the found pattern
     * @param x
     *          horizontal alignment of the pattern as double value
     * @param y
     *          vertical alignment of the pattern as double value
     */
    public synchronized void patternFound(String pattern, double x, double y) {
        for (int i = 0; i < this.pattaktion.length; i++) {
            // Go through all patterns and find the choosen one
            if (pattern.equals(pattaktion[i].getPattname())) {
                int richtung = this.erkannteAusrichtung(y, x);
                // Schould reakt on the found direccion
                if (pattaktion[i].getDireccion() == richtung
                        || pattaktion[i].getDireccion() == 5) {
                    if (null != pattaktion[i].getAudioNameArray()) {
                        String[] adatei = pattaktion[i].getAudioNameArray();
                        // Spiele alle Audiodateien zugeordnet zu dieser Aktion
                        for (int j = 0; j < adatei.length; j++) {
                            this.playSound(adatei[j]);
                        }
                    }
                    this.changeMotion(MotionModel.STOP_WR);
                    while (play.isPlaying()) {
                        waitTime(500);
                    }
                    this.changeMotion(pattaktion[i].getAction());
                    this.writeToLog("Found Pattern: " + pattern);
                    break;
                }
            }
        }
    }

    /**
     * Output of the GUI
     */
    public static void createAndShowGUI() {
        frame = new JFrame("Zebus Control Panel");
        frame.setResizable(false);
        frame.setSize(new Dimension(661, 480));
        // center frame on screen
        frame.setLocation(Toolkit.getDefaultToolkit().getScreenSize().width / 2
                - frame.getWidth() / 2,
                Toolkit.getDefaultToolkit().getScreenSize().height / 2
                        - frame.getHeight() / 2);
        ZCP contentPane = new ZCP();
        if (null != uro)
            frame.setIconImage(Toolkit.getDefaultToolkit().getImage(uro));
        contentPane.setOpaque(true);
        frame.setContentPane(contentPane);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing(WindowEvent e) {
                try {
                    cominterface.out("x");
                    cominterface.closeStreams();
                } catch (IOException e1) {
                    e1.printStackTrace();
                } catch (NullPointerException ne) {
                }
                cominterface = null;
                System.gc();
```

```
                System.exit(0);
            }
        });
    }

    /**
     * @param args
     *          common parameters of the main method
     */
    public static void main(String[] args) {
        JFrame.setDefaultLookAndFeelDecorated(true);
        // Schedule a job for the event-dispatching thread:
        // creating and showing this application's GUI.
        javax.swing.SwingUtilities.invokeLater(new Runnable() {
            public void run() {
                createAndShowGUI();
            }
        });
    }
}
```

E. Installation der verwendeten Software

Installationsanleitung der verwendeten Software

Die Installation der ARToolkit Bibliothek benötigt viele Komponenten, auf die die Bibliothek zugreifen muss, und mit der man das ARToolkit kompilieren kann. Es ist möglich als Betriebssystem Linux, Windows, MacOS und einige Andere zu benutzen. Diese Anleitung bezieht sich nur auf das Betriebssystem Windows. Desweiteren muss eine Kamera und deren Treiber vorinstalliert sein.

Vorbereitungen

Auf dem Rechner muss folgende Software installiert sein, um die verschiedenen Bibliotheken benutzen zu können (Reihenfolge bitte beachten):

- Cygwin (http:// www.cygwin.com)
- Microsoft Visual Studio 2005 (http://www.microsoft.com)
- JDK und JRE ab der Version 1.5 (http://java.sun.com)
- Java3D (http://java.sun.com)
- Eclipse 3.2 (http://www.eclipse.org)

Folgende Bibliotheken werden ebenfalls benötigt

- DSVideoLib-0.0.8b-win32 (http://sf.net/projects/artoolkit)
- GLUT (http://www.xmission.com/~nate/glut/glut-3.7.6-bin.zip)
- OpenVRML-0.14.3-win32 - (Optional) (http://sf.net/projects/artoolkit)

Installation der ARToolkit Bibliothek

1. Das ARToolkit muss von der folgenden Internetadresse bezogen werden: http://sf.net/projects/artoolkit
2. Die heruntergeladene Datei "ARToolKit.zip" muss in einen Ordner entpackt werden. Dieser Ordner wird weiterhin als {ARToolKit} bezeichnet.
3. Die Datei "DSVideoLib.zip" muss in den Ordner {ARToolKit} entpackt werden. Der Zielordner in dem die entpackten Dateien liegen sollen, muss „DSVL" benannt werden.
4. Die Dateien „DSVL.dll" und „DSVLd.dll" müssen von „{ARToolKit}\DSVL\bin" in das Verzeichnis „{ARToolKit}\bin" kopiert werden.
5. Die Bibliothek GLUT muss installiert werden, dabei sollte man die Instruktionen der beiliegenden „README.win" Datei innerhalb der „GLUT.zip" befolgen. Folgende Anleitung kann bei der Installation von GLUT behilflich sein: http://www.hitlabnz.org/forum/showpost.php?p=332&postcount=12.
6. Das Skript {ARToolKit}\Configure.win32.bat muss ausgeführt werden, um die Headerdatei include\AR\config.h zu erstellen.
7. Die Datei ARToolKit.sln oder ARToolkit.dsw kann dann mit „Visual Studio" geöffnet werden.
8. Das ARToolkit kann nun kompiliert werden.

Die VRML Renderer Bibliotheken und Beispiele (libARvrml & simpleVRML) sind optional:

9. Die Datei "OpenVRML.zip" muss in den Ordner {ARToolKit} entpackt werden.

10. Die Datei „js32.dll“ muss von „{ARToolKit}\OpenVRML\bin“ in das Verzeichnis “{ARToolKit}\bin“ kopiert werden.
11. In der Entwicklungsumgebung Visual Studio müssen die dem ARToolkit beiliegenden Projekte „libARvrml“ und „simpleVRML“ über den Konfigurationsmanager aktiviert werden.

Installation der JARToolkit Bibliothek

1. Das JARToolkit muss von folgender Internetadresse heruntergeladen werden: http://sourceforge.net/projects/jartoolkit/
2. Die Datei „jartoolkit.zip“ muss in einen Ordner entpackt werden.
3. Die entpackten Projekte müssen in das Arbeitsverzeichnis von „Eclipse“ importiert werden.
4. Der Klassenpfad in den Umgebungsvariablen von Windows muss einen Verweis auf den Ordner “bin” erhalten.
5. Eine der beiliegenden Demos kann aus „Eclipse“ heraus gestartet werden.

Installation der Plattformsoftware

1. Das Aksenboard kommt mit einer beiliegenden CD, die sämtliche Software zum Programmieren des Boards benötigt wird. Siehe Installationsanleitung auf der CD.
2. Die selbst geschriebene Plattformsoftware „Zebus“ befindet sich auf einer DVD, die dieser Arbeit beiliegt.
3. Der Ordner Zebus muss von der DVD in das Beispielverzeichnis des „AksenLib“-Ordners. Dort muss die beinhaltete Datei „zebus.c“ mit dem „make“-Befehl kompiliert werden.
4. Die daraus entstandene Datei „zebus.ihx“ muss mit dem „Flasher“ auf das Aksenboards übertragen werden.
5. Das Aksenboard wird zum starten der Software in den „Run“-Modus versetzt.

Installation der Steuerungssoftware

Die Steuerungssoftware „ARViewer“ liegt gepackt auf der beiliegenden DVD. Nach dem entpacken muss lediglich die Datei „start.bat“ ausgeführt werden. Sämtliche Voraussetzungen und vorausgehende Installationen müssen erfüllt und getätigt worden sein.

Damit die Steuerungssoftware auch Befehle an den Roboter über die serielle Schnittstelle senden kann, muss der Rechner durch ein serielles Kabel mit dem Aksenboard verbunden sein.

Die Steuerungssoftware startet nur wenn eine Kamera mit dem Rechner verbunden ist!

F. Übersicht der Arbeitsaufteilung

Übersicht Arbeitsverteilung

[AG = bearbeitet von André Gratz
BK = bearbeitet von Björn Kaiser]

Glossar

MARVIN Multi-prupose Aereal Robot Vehicle with Intelligent Navigation

A4 Aggregat 4

AIFF Audio Interchange File Format

BMBF Bundesministerium für Bildung, Wissenschaft, Forschung und Technologie

EADS European Aeronautic Defence and Space Company

EVA Evaluierungsexperiment Augsburg

GPS Global Positioning System

HAW Hochschule für Angewandte Wissenschaften in Hamburg

IGA Internationalen Gartenbauausstellung

JAR Java Archive

ORCA Observing Remote Controlled Airship

RAM Random Access Memory

RF Radio Frequency

RTOS Real Time Operating System

SIS Signal Intelligence Sensorpaket

SUN SUN Microsystems - Computerfirma ansässig im Silicon Valley

UAV Unmanned Aircraft Vehicle

V2 Vergeltungswaffe 2

WAV Containerformat für Audiodaten

WLAN Wireless Local Area Network

Zebus Zeppelin Kubus

Index

Versicherung über Selbstständigkeit

Hiermit versichern wir, dass wir die vorliegende Arbeit im Sinne der Prüfungsordnung nach §24(5) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt haben.

Hamburg, 23. August 2007
Ort, Datum

Unterschrift

Unterschrift