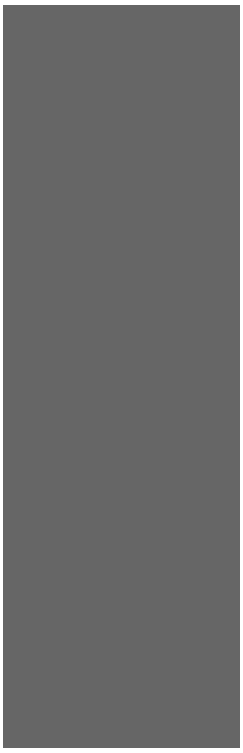
GMD-Report 126

GMD –
Forschungszentrum
Informationstechnik
GmbH

Hartmut Surmann, Kai Lingemann,
Andreas Nüchter, Joachim Hertzberg



Aufbau eines 3D-Laserscanners für autonome mobile Roboter

März 2001

GMD – Forschungszentrum Informationstechnik GmBH
Schloß Birlinghoven
D-53753 Sankt Augustin
Germany
Telefon +49-2241-14-0
Telefax +49-2241-14-2618
http://www.gmd.de

Anschrift der Verfasser/Address of the authors:
Dr. Hartmut Surmann
Kai Lingemann
Andreas Nüchter
Dr. Joachim Hertzberg
Institute für Autonome intelligente Systeme
GMD - Forschungszentrum Informationstechnik GmbH
D-53754 Sankt Augustin
E-Mail: surmann@gmd.de

Die Deutsche Bibiothek - CIP-Einheitsaufnahme
Aufbau eines 3D-Laserscanners für autonome mobile Roboter /
GMD - Forschungszentrum Informationstechnik GmbH,
Hartmut Surmann . . .- Sankt Augustin: GMD - Forschungszentrum
Informationstechnik, 2001
(GMD-Report; 126)
ISBN 3-88457-974-6

ISSN 1435-2702
ISBN 3-88457-974-6

Zusammenfassung

Diese Ausarbeitung stellt das Konzept und die Realisierung eines 3D-Laserscanners vor. Ziel der Realisierung war es, das Dreieck aus Kosten, Geschwindigkeit und Qualität so zu optimieren, daß der 3D-Scanner auf autonomen mobilen Robotern sinnvoll zur Exploration und Navigation eingesetzt werden kann. Ein auf autonomen mobile Fahrzeugen häufig verwendeter 2D-Laserscanner wurde dazu mittels einer selbst konstruierten Aufhängung und eines Servomotors aufgerüstet. Mittels des auf einem Standardrechner laufenden Echtzeitbetriebssystem RT-Linux steuert der Servomotor den 3D Laserscanner direkt an. Unterschiedliche Online-Algorithmen zur Linienerkennung und Flächendetektion bilden die Software-Basis des 3D Scanners. Offline-Algorithmen zur Objektsegmentierung und Erkennung sowie ein 3D-Visualisierungsprogramm runden das Softwarepaket ab.

Schlagwörter: 3D-Laserscanner, autonome mobile Roboter, Sensorfusion, Linienerkennung, Objekterkennung, Objektsegmentierung

Abstract

This report presents the concept and implementation of a 3D laser range finder. The objective of the implementation was to optimize the triangle between costs, speed and quality in such a way that the 3D laser range finder can be used for the exploration and navigation for autonomous mobile robots. A 2D laser range finder usually used by autonomous mobile robots was extended by a self designed suspension and a servo motor. The servo of the new 3D laser range finder is controlled from a computer running the real-time operating system RT-Linux. Different online algorithms for line and surface detection are the software base of the new 3d laser range finder. Offline algorithms for the object segmenting and recognition as well as a 3D viewer and a graphical user interface are extending the software package.

Keywords: 3D laser range finder, autonomous mobile robots, sensor fusion, line detection, object detection, object segmentation

Danksagung

Für ihre Hilfe bei dem Projekt „3D Laserscanner“ möchten wir uns an dieser Stelle bei Adriana Arghir und Nicolas Andree bedanken.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Wesentliche Teile dieser Arbeit entstanden im Rahmen eines Programmierpraktikums „3D-Laserscanner“. Aufgabe war es, einen handelsüblichen 2D-Laserscanner dahingehend zu modifizieren, daß Szenen in 3 Dimensionen aufgenommen werden. Der entwickelte 3D-Laserscanner soll speziell zur Navigation und Exploration auf autonomen mobilen Robotern eingesetzt werden.

Wichtige Fragestellungen in der Robotik sind die Kartierung und Navigation der Umgebung sowie anwendungsspezifische Aufgaben der jeweiligen Roboter, beispielsweise Andock- und Einparkaufgaben. Für solche autonomen Systeme braucht man *präzise*, *schnelle* und *preisgünstige* Sensoren. Heute verwenden viele Roboter 2D-Laserscanner. Die Erstellung von 2D-Karten und Navigation ist damit in Grundzügen möglich [Sur95]. Es besteht jedoch nicht die Möglichkeit, Hindernisse auf Grundlage eines 2D-Scans vollständig zu erfassen, da diese beispielsweise überstehende Kanten aufweisen können. Objekterkennung ist ebenfalls nicht ohne den Einsatz weiterer Sensoren wie beispielsweise Kameras möglich. Ein 3D-Laserscanner bietet für Probleme dieser Art eine echte Alternative.

Der in dieser Arbeit vorgestellte Ansatz verwendet einen 2D-Laserscanner, der auf einer horizontalen, senkrecht zur Hauptblickrichtung des Scanners verlaufenden Drehachse montiert wird. Dadurch ist es möglich, den Scanner wie einen herkömlichen 2D-Laserscanner zu betreiben. Ferner können vorhandene Algorithmen weiterhin zur Roboter-Lokalisation eingesetzt werden [Arr96]. Zusätzlich ist es nun möglich, 3D-Szenen aufzunehmen. Dabei wird der Laserscanner um einen Winkel von 90 Grad in der horizontalen Drehachse gedreht. Ein handelsüblichen Servomotor aus dem Modellbaubereich dreht den 2D-Scanner.

Die *Präzision* des zu realisierenden 3D-Senosrs soll eine Mindestgenauigkeit von 5cm besitzen. In jeder Scanebene ist die Genauigkeit durch die Genauigkeit des Lasers bestimmt, die bei dem in dieser Arbeit verwendeten Laser bei etwa 4cm liegt. Die vertikale Genauigkeit bestimmt sich durch die Servogenauigkeit und dem Ansteuerungsverfahren des Servos. Die hier erreichte Genauigkeit liegt bei etwa 0.5 Grad. Für die Ansteuerung des Servos wird das Echtzeitbetriebsystem RT-Linux verwendet. Der 3D-Laserscanner besitzt eine Auflösung von bis zu 113400 Punkten

(420×270) bei einem Öffnungswinkel von $150° \times 90°$. Damit können Objekte im Nahbereich ab einer Größe von 5cm $\times$ 5cm $\times$ 5cm erkannt werden.

Neben der Qualität ist die *Geschwindigkeit* ein entscheidendes Kriterium für den Einsatz von Sensoren auf mobilen Platformen. Die Zeit, die für eine Aufnahme einer 3D-Szene benötigt wird, ist von der Geschwindigkeit des Lasers abhängig sowie der gewählten Auflösung. Die Zeit für einen kompletten Scan von $150° \times 90°$ beträgt – je nach Auflösung – zwischen 1 und 12 Sekunden, wobei der Scanner alle 30ms einen 2D-Scan aufnimmt.

Letztendlich bestimmt auch der Preis über den Einsatz von Sensoren auf mobilen Fahrzeugen. Der 3D-Scanner ist sehr *preisgünstig*, da ein vorhandener 2D-Laserscanner um einen Modellbauservo und eine Aufhängung erweitert wurde. Der eingesetzte Servo der Firma Multiplex kostet €110,—.

Werden die autonomen Roboter zur Vermessung von Gebäuden und zur Kartengenerierung eingesetzt, so ist eine realistische Visualisierung erwünscht. Zu diesem Zweck wurde eine Digitalkamera an den Laserscanner gekoppelt. Aus dem während des 3D-Scans aufgenommenen Kamerabildern werden Texturen für die 3D-Szene extrahiert.

Online-Algorithmen zur Erkennung von Linien und Flächen bilden die Software-Basis für die anschließende Objektsegmentierung und Erkennung. Die Visualisierung erfolgt durch eine seperate Software auf der Basis der OpenGL-Bibliothek.

Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich in 7 Kapitel:

Kapitel 1 die vorliegende Einleitung.

Kapitel 2 gibt einen Überblick über den aktuellen Stand der Technik, der einen direkten Vergleich der vorliegenden Arbeit mit bereits auf dem Markt erhältlichen Produkten erlaubt. Es folgen eine kurze Darstellung der verwendeten Hardware, insbesondere der von uns designten Drehvorrichtung, des verwendeten Laserscanners und der Kamera.

Kapitel 3 beschäftigt sich mit den zeitkritischen Realisierungen. Dies sind zum einen das unterliegende Betriebsystem RT-Linux, zum anderen die implementierten Online-Algorithen wie die Linienerkennung und die Flächensegmentierung.

Kapitel 4 erläutert die verwendeten Offline-Algorithmen, insbesondere zur Objekt-Segmentierung.

Kapitel 5 stellt den internen Ablauf der Scanner-Applikation vor, die parallele Architektur, die Klassenhierarchie sowie das JAVA-Interface zur benutzerfreundlichen Bedienung des Programmes.

Kapitel 6 stellt die Visualisierung der 3D-Scans vor und geht nähter auf die Funktionalität des OpenGL-Programmes ein, gefolgt von einigen Beispielbildern, aufgenommen bei einem Gang durch das Gebäude.

Kapitel 7 faßt die Ergebnisse der Arbeit zusammen und liefert einen Ausblick auf zukünftige Arbeiten.

Kapitel 2

3D-Laserscanner

Dieses Kapitel gibt einen Überblick über den aktuellen Stand der Technik, der einen direkten Vergleich der vorliegenden Arbeit mit bereits auf dem Markt erhältlichen Produkten erlaubt. Es folgen eine kurze Darstellung der verwendeten Hardware, insbesondere der von uns entwickelten Drehvorrichtung, des verwendeten Laserscanners und der Kamera.

2.1 Stand der Technik

In diesem Abschnitt werden verschiedene 3D-Laserscanner und Verfahren zur Vermessung von 3D-Szenen vorgestellt. Neben Laserscannern gibt es weitere Möglichkeiten, ein dreidimensionales Abbild eines Objektes oder der Umgebung zu erstellen (z.B. Ultraschallsysteme, Stereokameras) [Bor98].

Kommerzielle 3D-Laserscanner lassen sich entsprechend ihrer Funktionsweise in zwei Gruppen einteilen:

- Zum einen gibt es *punkt-* beziehungsweise *streifenbasierte* Laserscanner. Hierbei wird ein Laserpunkt oder Laserstreifen auf ein Objekt projiziert. Mehrere Sensoren detektieren diesen Punkt bzw. Streifen. Mittels Trianguliation wird der Abstand zum Objekt bestimmt oder aus der Verformung des Musters die Geometrie erschlossen [Her99].
- Zum anderen gibt es *Lichlaufzeitmessungs-basierte* Laserscanner. Dabei wird ein gepulster Laserstrahl zum Objekt gesenden und dort diffus reflektiert. Eine Empfangsdiode mißt das reflektierte Licht und bestimmt aus der Lichtlaufzeit den Abstand des Objektes.

Die Auflösung bei Lichtlaufzeitmessungen ist gegenüber dem ersten Verfahren in der Regel geringer, jedoch sind diese schneller.

Desweiteren kann man die erhältlichen 3D-Laserscanner nach Ihrer Größe einteilen:

- Zum einen gibt es eine Reihe von – meist portablen – Scannern, die in der Lage sind, kleine Objekte einzuscannen.

- Scanner für große Objekte sind groß, schwer und unbeweglich [Bor98].

Für den Einsatz in der Robotik sind beide Arten ungeeignet.

2.1.1 3D-Scanner zum Scannen kleiner Objekte von Conrad

Ein Scanner, der sehr kleine Objekte bis zu einer Größe von 20cm $\times$ 20cm $\times$ 20cm einzuscannen vermag, gibt es bei dem großen deutschen Versandhaus Conrad zu kaufen [Con01].

Dieser optische Scanner vermißt Objekte, welche sich auf einem Drehteller befinden, nach dem Triangulationsprinzip. Er besitzt eine Auflösung von rund 0.5 Grad und bietet zusätzlich zwei Kameras, um die gescannten Objekte mit Texturen zu belegen. Der Scanner kostet €766,—.

2.1.2 Minolta 3D Scanner

Minolta produziert mit dem VI-700 einen 3D-Scanner (streifenabtastender Triangulationsscanner) [URLMin]. Objekte bis zu einer Größe von 1.1m $\times$ 1.1m $\times$ 1.5m werden vermessen. Dabei werden in 0.6 Sekunden 200 $\times$ 200 Punkte eingescannt. Werbezitat: „Wird das Objekt von mehreren Seiten gescannt (z.B. auf dem optionalen Rotary-Table), so können diese Einzelscans auf einfachste Art und Weise zu einem vollständigen Objekt zusammengesetzt werden. Die eingebaute Digitalkamera erstellt zusätzlich zum 3D-Scan ein Bild, welches auf Wunsch als Textur über das Objekt gelegt werden kann.“ [URLMin]. Ein solches portables Gerät kostet etwa €25.550,—.

2.1.3 Cyberwave 3D Scanner

Die Firma Cyberwave stellt 3D-Scanner für den High-End Markt her [Bor98]. Dabei können Objekte bis zu einer Größe von zirka 2m $\times$ 1.5m $\times$ 1.5m eingescannt werden. Das zugrundeliegende automatische streifenbasierte optische Scannverfahren besitzt Genauigkeiten bis zu 300μm. Um ein 3D Modell zu erhalten, wird entweder das Objekt rotiert (Cyberwave WB4) oder die ganze Scannaperatur bewegt (Cyberware Model 15). Die Scanner sind nicht oder nur schlecht portabel. Kosten: $>$ €100.000,—.

2.1.4 Autoscan

Autoscan ist ein 3D Scansystem, welches auf punktbasiertem optischen Scannen beruht [Bor98]. Das System besteht aus einem Laserpointer, zwei Videokameras und einem Real-Time Bildprozessor. Der Laserpointer wird manuell, d.h. von Hand, bewegt und markiert einen Punkt auf dem zu scannenden Objekt. Über die zwei Kameras wird dann via Triangulation die Entfernung zu dem Punkt berechnet. Die Genauigkeit variiert mit der Breite der Stereobasis, erreicht aber Werte bis zu 0.1mm. Nachteilig ist, daß dieses Verfahren extrem hohen Rechenaufwand besitzt, welches bisher (1998) nur Spezial-Hardware zu lösen vermochte.

2.1.5 Zeitmessungsbasierte 3D-Laserscanner

Zeitmessungsbasierte 3D-Laserscanner sind aus der Sicherheitstechnik stammende modifizierte 2D-Laserscanner. Um auch damit dreidimensional scannen zu können muß entweder die Apparatur beziehungsweise das Objekt bewegt werden. In der Literatur sind zwei Apparaturen bekannt, die 3D-Laserscans ermöglichen und dabei den 2D-Laserscanner drehen. Zum einen ist das der 3D-Scanner auf Daimler-Chrysler Roboter (Neuros Projekt) [Jen99] mit einer Drehvorrichtung von Amtec und zum anderen der SICK-Laser Range Finder auf MAVERICK [Wal00], der ebenfalls ein Amtec Modul zur Drehung um die vertikale Achse benutzt [URLAmt, URLSic].

Die Genauigkeit hängt von dem Laserscanner und dem externen Drehmodul ab, das Auflösungsvermögen nimmt mit der Entfernung ab. Hardwarekosten für einen solchen Aufbau sind etwa €6.000,—. Mit diesem Verfahren können sehr große Objekte gescannt werden, einschließlich ganzer Räume. Deshalb eignet sich dieses Verfahren in der Robotik sehr gut.

2.1.6 3D Objekterfassung auf Silizium

Ein neuer CMOS-Bildsensor, der dreidimensionale Objekte erfaßt, wurde von Siemens und der FhG entwickelt [URLSie]. Im Gegensatz zu dem bisher vorgestellten Verfahren mit Lichtlaufzeitmessung muß hier kein Spiegel mehr gedreht werden. Der Chip basiert auf einem sogenannten MDSI (Multiple Double Short Time Integration) Verfahren. Zitat: „[Das] gesammte auszumessende Objekt wird mit Laserimpulsen niedriger Leistung beleuchtet und das zurückkommende Licht durch CMOS-Bildwandler mit extrem kurzen Integrationszeiten erfaßt.“ [URLSie]. Das Verfahren befindet sich zur Zeit noch in der Entwicklung, erreicht eine Auflösungen von bis zu 1000 Objektpunkten (30×30) und Genauigkeiten bis 1cm.

2.1.7 Callidus

Callidus ist ein 3D-Lasermeßsystem der Callidus Precision Systems GmbH, welches die Vermessung von Innenräumen ermöglicht [URLCal]. Auch Callidus arbeitet mit Pulslaufzeitmessung. Der Laserscanner läßt sich horizontal um 360° drehen und überdeckt vertikal einen Bereich von 180° mit einer Winkelauflösung von bis zu 0.25°.

Der Laserscanner wird statisch auf einem Stativ positioniert und scannt den gesamten von dieser Position aus einsehbaren Raum. Die Zeit für einen Scan liegt zwischen 3 Minuten und mehreren Stunden. Der Preis einschließlich Auswertungssoftware liegt bei ca. €50.000,—.

2.1.8 3D Scannen durch Roboterbewegung

Eine weitere interessante Möglichkeit, insbesondere größer Objekte wie Räume, etc. dreidimensional zu scannen, ist, einen 2D Laserscanner (vgl. 2.2.2) mit „Blickrichtung“ nach oben auf einer mobilen Roboterplattform zu befestigen[Thr00]. Durch Drehbewegungen des Roboters wird eine

Abfolge von 2D-Daten gesammelt, die zu einem 3D-Modell kombiniert werden können. Die Drehung des Roboters ersetzt somit einen externen Motor. Zur Erstellung eines 360° Scans ist eine Drehung der Plattform um 180° notwendig. [URLThr].

Die Meßgenauigkeit dieses Verfahrens hängt wesentlich von der genauen Kenntnis der Roboterposition ab. Der Prozeß Datenaufnahme ist also an den Prozeß der Bewegung gekoppelt. Über die erreichten Genauigkeiten bei den auf diesem Prinzip erstellten 3D-Szenen liegen keine genauen veröffentlichten Informationen vor. Schätzungsweise dürfte die Genauigkeit im Bereich von ungefähr 20cm liegen.

Fazit

Da die zwei 3D-Laserscanner von den Forschungsgruppen Daimler-Chrysler und der Uni Lübeck nicht komerziell verfügbar sind, sind zur Zeit auf dem Markt keine preisgünstigen mobilen 3D-Laserscanner für größere Objekte erhältlich, die in der Robotik einsetzbar wären.

2.2 Aufbau des realisierten 3D-Scanners

Grundlage des im folgenden dargestellten 3D-Laserscanners ist ein 2D-Laserscanner der Firma Schmersal [URLSch], der drehbar gelagert wurde (Abbildung 2.1).

Ein entscheidendes Kriterium für den Anschluß von Hardware an den Rechner ist die Auswahl der benötigten Schnittstellen. Es werden lediglich *Standardschnittstellen* des Computers benutzt, um die Peripheriegeräte anzuschließen.

Schnittstelle	angeschlossenes Gerät
serielle Schnittstelle (COM1)	Schmersal Laserscanner
parallele Schnittstelle (LPT1)	Servomotor
Universal Serial Bus (USB)	Kamera

Tabelle 2.1: Schnittstellen des 3D-Laserscanners

Diese Schnittstellen sind heutzutage auf portablen Rechnern vorhanden, wodurch die Apparatur universell und flexibel auf mobilen Roboterplattformen einsetzbar ist, vgl. Abbildung 2.2.

2.2.1 Drehvorrichtung

Bedingungen an den Aufbau stellen sich durch die technischen Daten der verwendeten Hardware. Der Laserscanner wiegt circa 3.5kg bei Abmessungen von 155mm $\times$ 185mm $\times$ 156mm. Um ein solches Objekt drehen zu können, müssen Rotations- und Reibungskräfte überwunden werden. Der verwendete Modellbauservo hat ein maximales Drehmomente von 210Ncm.

Zur Realisierung der Drehvorrichtung wurde der Laserscanner auf zwei „Füße“ gestellt. Die Drehung erfolgt senkrecht zur Blickrichtung (Abbildung 2.1). Da das zur Verfügung stehende Drehmoment begrenzt ist, mußte das Trägheitsmoment minimiert werden.

Abbildung 2.1: Der Aufbau des 3D Laserscanners. Die Drehachse verläuft horizontal

Abbildung 2.2: Montage auf dem Roboter [URLHp]

Es gilt:

$$\begin{array}{lcccc} \text{Drehmoment} & = & \text{Trägheitsmoment} & \cdot & \text{Winkelbeschleunigung} \\ \vec{M} & = & J_s & \cdot & \dfrac{\mathrm{d}\vec{\omega}}{\mathrm{d}t} \qquad \left(= \vec{r} \times \vec{F}\right) \end{array} \tag{2.1}$$

mit dem Trägheitsmoment nach dem Steinerschen Satz

$$J_s = J_a + r_d \cdot m^2 \quad (2.2)$$
$$= \int r \, \mathrm{d}m + r_d \cdot m^2, \quad (2.3)$$

wobei J_a das Trägheitsmoment bei Drehung um eine Schwerpunktachse ist, r bezeichnet den Abstand eines Massepunktes vom Schwerpunkt und r_d ist der Abstand der Drehachse von einer parallelen Achse, die durch den Schwerpunkt geht. Somit ergibt sich, daß sowohl die Masse m die gedreht werden muß als auch der Abstand von der optimalen Drehachse (Schwerpunktdrehachse) r_d möglichst klein gehalten werden müssen.

Zur Bestimmung der optimalen Drehachse wird der zu drehende Körper auf zwei parallel verlaufenden horizontalen Achsen aufgehängt und ausgependeld. Dabei wird jeweils das Lot gefällt. Der Schnittpunkt der beiden Lote ergibt die optimale Drehachse des Körpers.

Eine weitere Realisierungsmöglichkeit mit horizontaler Drehachse

Eine alternative Möglichkeit, einen Standard Sicherheits-Laserscanner wie den LSS 300 zu einem 3D-Laserscanner umzubauen, ist die Montage eines 2D-Laserscanners auf eine drehbare Platte. Dabei liegt die Drehachse ebenfalls horizontal und zwar in Richtung der „Blickrichtung“. Dies könnte zum Beispiel durch ein Modul wie in Abbildung 2.3 dargestellt geschehen.

Der Vorteil eines solchen Aufbaus ist, daß man alles vor dem Laser Befindliche erfaßt und nicht nur einen Bereich von 90 $\times$ 180 Grad. Nachteile für diese Lösung sind neben den höheren Kosten, daß ein 3D-Scan die doppelte Zeit für die Aufnahme benötigen würde, da man hierbei darauf angewiesen wäre, daß der Laser um volle 180 Grad gedreht wird.

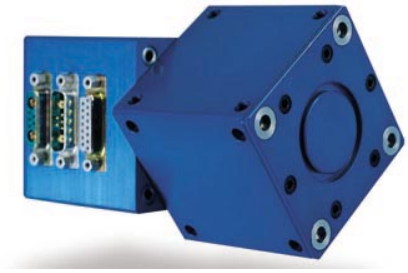


Abbildung 2.3: Das Drehmodul

2.2.2 Sicherheits-Laserscanner LSS 300

Zur Skizze der wesentlichen technischen Merkmale des verwendeten 2D-Scanners zitieren wir aus der Produktbeschreibung [URLSch]:

„Die Sicherheits-Laserscanner der Reihe LSS 300 detektieren berührungslos eine 180 Grad-Fläche und kann so zur horizontalen Absicherung eines Gefahrenbereichs vor oder in Anlagen, wie

z.B. Rohrbiegemaschinen, Industrieroboter oder fahrerlose Transportmittel eingesetzt werden. Die wartungsfreien Sicherheits-Laserscanner der Reihe LSS 300 senden einen ungefährlichen, nicht sichtbaren Laserstrahl aus. Durch einen sich drehenden Spiegel wird eine 180 Grad-Flächendetektion erwirkt. Der Laserstrahl wird von Objekten im Scanbereich diffus reflektiert und wieder im Laserscanner empfangen. Die interne Auswertung errechnet durch Lichtlaufzeitmessung und aus den korrespondierenden Winkelinformationen die genaue Entfernung und Position der jeweiligen Person oder des Gegenstandes. Befindet sich diese Person oder der Gegenstand innerhalb des vorher über die benutzerfreundliche, interaktive Software programmierten Schutz- oder Warnfeldes, werden die Ausgänge des Sicherheits-Laserscanners geschaltet.

Abbildung 2.4: Der Laserscanner

Der LSS 300 erfüllt die Schutzart IP 67 und entspricht der Steuerungskategorie 3 nach EN 954-1. Das Modell LSS 300-2 kann bis zu 2 Schutzzonen (innerhalb 4m) und bis zu 2 Warnzonen (bis 15m) absichern. Die Ansprechzeit liegt unter 60ms, die Auflösung beträgt 50mm in 4m Entfernung. Anlauf- und Wiederanlaufsperre können wie die Schutzkontrolle an- und ausgeschaltet werden.“ [URLSch]

Abbildung 2.5: Die Mustek MDC800

2.2.3 Kamera

Zur realitätsnahen Visualisierung der 3D-Szene wird die USB-Kamera MDC800 (siehe Abbildung 2.5) von Mustek eingesetzt. Während des Scans werden Photos aufgenommen und daraus Texturen extrahiert. Diese Texturen werden später mit den 3D-Elemente verknüpft.

Die Mustek MDC800

Die Kamera besitzt eine Auflösung von maximal 1012 × 768 Pixel, 4MB internen Speicher, einen Öffnungswickel von 55 Grad und einen fixierten Fokus von 70cm bis unendlich.

Zum Betrieb der Kamera ist ein Kerneltreiber unbedingt notwendig. Wir verwenden den mdc800-Kerneltreiber von Henning Zabel [URLGph]. Der Treiber wurde leicht modifiziert, um den zeitkritischen Anforderungen gerecht zu werden. Dieser Kerneltreiber stellt den Anwendungsprogrammen `read/write` Operationen auf des Gerät `/dev/mustek` zur Vefügung, so daß diese dann das Protokoll implementieren können.

Für die Implementierung des USB-Protokolls wird das Linux-Programm `gphoto` benutzt. Dieses Programm wird aus der Scannerapplikation heraus aufgerufen. Die Kamera benötigt ungefähr 9 Sekunden, um ein Photo aufzunehmen (inklusive der Downloadzeit).

2.2.4 Technische Daten des entwickelten 3D-Scanners

Größe (B × H × T)	350mm × 240mm × 240mm
Gewicht	4.5kg
Betriebsspannung Laserscanner	24V
Leistungsaufnahme Laserscanner	20W (max.)
Betriebsspannung Servomotor	6V
Leistungsaufnahme Servomotor	
maximal	12W
durchschnittlich	3W
Ruhe	1.2W
Drehmoment Servo	210Ncm
Anschluß Laserscanner	RS232/RS422
Anschluß Servomotor	LPT
scannbarer Bereich	150° × 90°
maximale Auflösung	113400 Punkte
maximale Genauigkeit	vertikal: 0.5°
	horizontal: 5cm
Reichweite	60m

min. Objektremission	1.8% (diffus)
max. Objektremission	keine Beschränkung
Lichtquelle (Wellenlänge)	Laserdiode (905nm)
Laserschutzklasse	1 nach EN
Scanngeschwindigkeit	4-12s, ja nach Auflösung

Kapitel 3

Echtzeitsteuerung und Online-Algorithmen

In diesem Kapitel wird die Umsetzung der zeitkritischen Prozesse beschrieben. Dies sind zum einen das unterliegende Betriebsystem RT-Linux, zum anderen die implementierten Online-Algorithmen wie die Linienerkennung und die Flächensegmentierung.

3.1 Real-Time Linux

Real-Time Linux ist ein Betriebssystem, bei dem ein kleiner Echtzeitkernel und der Linux Kernel nebeneinander existieren. Es soll erreicht werden, daß die hochentwickelten Aufgaben eines Standardbetriebssystems für Anwendungen zur Verfügung gestellt werden, ohne auf Echtzeitaufgaben mit definierten, kleinen Latenzen zu verzichten. Bisher waren Echtzeitbetriebssysteme kleine, einfache Programme, die nur wenige Bibliotheken dem Anwender zur Verfügung gestellt haben.

Mittlerweile ist es aber notwendig geworden, daß auch Echtzeitbetriebssysteme umfangreiche Netzwerk-Bibliotheken, Graphische Benutzeroberflächen, Datei– und Datenmanipulationsroutinen zur Verfügung stellen müssen. Eine Möglichkeit ist es, diese Features zu den existierenden Echtzeitbetriebssystemen hinzuzufügen. Dies wurde beispielsweise bei den Betriebssystemen VXworks und QNX gemacht. Eine andere Herangehensweise ist, einen bestehenden Betriebssystemkernel dahingehend zu modifizieren, daß dieser Echtzeitaufgaben bewältigt. Ein solcher Ansatz wird von den Entwicklern von RT-IX (Modcomp) verfolgt. RT-IX ist eine UNIX System V Implementation, die auch harte Echtzeitaufgaben bewältigen kann [Kuh98].

Real-Time Linux basiert auf einer dritten Herangehensweise. Ein kleiner Echtzeitkernel läßt ein Nicht-Echtzeitbetriebssystem als einen Prozeß mit niedrigster Priorität laufen. Dabei wird eine virtuelle Maschine für das eingebettete Betriebssystem benutzt.

Real-Time Linux behandelt alle Interrupts zuerst und gibt diese dann an den Linux Betriebssystem-

Prozeß weiter, wenn diese Ereignisse nicht von Echtzeit-Prozessen behandelt werden. Durch einen solchen Aufbau sind minimale Änderungen am Linux Kernel nötig, da der Interrupt Handler auf den Echtzeitkernel abgestimmt werden muß. Der Real-Time Linux Kernel selbst ist ein ladbares Modul und kann bei Bedarf eingefügt und entfernt werden. Dieser Kernel stellt hauptsächlich die Schicht zwischen dem Linux Kernel und der Hardware-Interruptebene dar.

Real-Time Linux mutet anfänglich sehr spartanisch an, denn es liegt der Gedanke zugrunde, daß die Prinzipien von harten Real-Time Anwendungen in der Regel nicht vereinbar sind mit komplexer Synchronisation, dynamischer Resourcenverwaltung und allem anderen, was zeitliche Mehrkosten verursacht. Daher werden in der Standardkonfiguration nur sehr einfache Konstrukte zur Implementierung von Prozessen zur Verfügung gestellt, beispielsweise ein einfacher fixed priority scheduler, nur statisch allozierter Speicher und und kein Schutz des Adreßraums. Dennoch kann man mit den gegebenen Mitteln sehr effizient arbeiten, da alle Nicht-Echtzeitkomponenten unter Linux laufen und somit dortige Bibliotheken benutzen können.

Real-Time Programme bzw. Prozesse werden als ladbare Module eingebunden, was das Echtzeitbetriebssystem erweiterbar und einfach zu modifizieren macht. Die Programme werden mit den Standard-Linux-Werkzeugen erzeugt (GNU C compiler). Oftmals übernehmen andere, Nicht-Real-Time Programme die weitere Datenverarbeitung. Da Real-Time Linux keinerlei Mechanismen zum Schutz gegen Überladung der Hardware besitzt, ist es möglich, daß die Real-Time Prozesse die CPU blockieren. In diesem Fall bekommt der Linux Kernel nicht die Chance, eine Zeitscheibe zu bekommen, da er die niedrigste Priorität hat.

Da Real-Time Linux OPENSOURCE ist und da die Module von Real-Time Linux ladbar und damit jederzeit austauschbar, sind, fand eine breite Entwicklung von Zusatzmodulen statt. Als Zusatzmodule stehen zur Zeit alternative Scheduler (z.B. rate-monotonic scheduler), ein Semaphore Modul, IPCs (interprocess communication Mechanismen) und etliche Real-Time device driver zur Verfügung.

Real-Time und Linux-User Prozesse kommunizieren über lock-freie Queues (FIFOs) und shared memory. Anwendungsprogrammierern stellen sich die FIFOs als standard character devices dar, auf die mittels POSIX `read/write/open/ioctl` zugegriffen wird.

Es hat sich gezeigt, daß Real-Time Linux den Anforderungen eines Echtzeitbetriebssystems sehr nahekommt. Die Worst-Case interrupt latency auf einem 486/33MHz PC liegt unter 30μs ($30 \cdot 10^{-6}$s) und somit nahe dem Hardwarelimit [RTL98].

3.1.1 Installation von Real-Time Linux

Von Real-Time Linux gibt es verschiedene Versionen. Etliche kommerzielle Hersteller bieten ihre Versionen von dem Betriebssystem Real-Time Linux an. Wir beziehen uns im folgenden auf die frei erhältliche Version dieses Betriebssystems, siehe [URLFsm].

Real-Time Linux gibt es bereits in der 3. Version. Mit der ersten Version wurde das Konzept implementiert und getestet. In Version 2 wurden etliche Features und die ganze API überarbeitet

sowie mit dem Betriebssystem POSIX konform gemacht. Es wird Real-Time Linux Version 2.2a verwendet, die auf dem letzten stabil laufendem Linux Kernel basiert (Kernel Version 2.2.14) (Stand: Juni 2000). Die neue Version 3 basiert auf den noch nicht freigegebenen Kernel der 2.4er Serie.

Die Tabelle 3.1 zeigt die Real-Time Linux Versionen vs. den Linux Kernel Versionen.

Linux Kernel Nummer	Real-Time Patch Version
2.0.27	0.5
2.2.14	2.2
2.2.14	2.2a
2.4.pre-test0	3

Tabelle 3.1: Real-Time Linux Patchliste

Zur Installation benötigt man nur ein lauffähiges Linux. Man lädt sich einen frischen, d.h. ungepatchten Kernel und den Real-Time Linux Patch. Alternativ dazu kann man sich auch die vor-gepatchten Kernelquellen besorgen. Danach folge man den Anweisungen der Installations-Dateien.

Problematisch ist die Installation, wenn man andere Kernelversionen als die zu Real-Time Linux gehörenden verwenden möchte. Der Real-Time Patch läßt sich i.A. nur auf die Kernelversion anwenden, für die er geschrieben wurde. Für den Fall, daß andere Patches angewendet werden müssen, sollte man diese auf den Real-Time Quellcode anwenden. So ist es möglich, ein Real-Time fähiges Betriebssystem und Unterstützung mit USB (Universal Serial Bus) verbinden zu können.

3.1.2 API – die Grundfunktionen

Im folgenden sind die wichtigsten in der Scanner-Software verwendeten Real-Time Interfacefunktionen vorgestellt. Dabei wird die neue POSIX-konforme API V2 verwendet.

Real-Time Scheduler

Die Aufrufe zur Prozeß-Verwaltung sind in der ab Version 2 dem Standard `pthread` angeglichen wurden. Für einen Real-Time-Prozeß steht folgende Struktur zur Verfügung.

```
struct rtl_thread_struct {
        int *stack;     /* hardcoded */
        int uses_fp;
        enum rtl_task_states state;
        int *stack_bottom;
        rtl_sched_param sched_param;
```

```
        struct rtl_thread_struct *next;
        RTL_FPU_CONTEXT fpu_regs;
        int cpu;
        hrtime_t resume_time;
        hrtime_t period;
        void *retval;
        int pending_signals;
        struct tq_struct free_task;
        void *user[4];
        int errno_val;
        int pad [64];
};

typedef struct rtl_thread_struct RTL_THREAD_STRUCT;
typedef RTL_THREAD_STRUCT *pthread_t;
```

Es genügt daher, einen Thread wie folgt zu definieren.

```
pthread_t REALTIME_TASK;
```

Eine Funktion (*Prozeß*) kann nun diesem Thread zugeordnet und gestartet werden:

```
int pthread_create(pthread_t  *thread,
                   pthread_attr_t *attr,
                   void *(*start_routine)(void *),
                   (void *arg));
```

Nun muß der Prozeß periodisch gemacht werden. Dies geschieht mittels folgender nicht-posix konformer (erkennbar an dem Postfix „_np“) Funktion

```
int pthread_make_periodic_np(pthread_t  thread,
                             hrtime_t start_time,
                             hrtime_t period);
```

Innerhalb des Prozesses lassen sich die Parameter des Threads (wie beispielsweise die Periode `REALTIME_TASK->period`) beeinflussen. Der Prozeß läßt sich bis zu dieser Periode mittels der Funktion

```
int pthread_wait_np(void);
```

explizit schlafen legen, die Steuerung wird an andere Prozesse zurückgeben. Prozesse mit der Priorität 1 haben die höchste Priorität, der normale Linuxkernel wird mit der niedrigsten betrieben.

Zeit-Verwaltung

Für Zeiten in Real-Time Linux wird die Datenstruktur `hrtime_t` verwendet. Dies ist ein 64-Bit Integer. Die aktuelle Auflösung der Zeit (in Prozessorticks) durch diese Variable ist hardwa-

reabhängig[1], es wird aber garantiert, daß die Auflösungsgenauigkeit unter 1μs (Mikrosekunde) liegt. Das Symbol `HRTIME_INFINITY` repräsentiert den maximalen Wert und wird niemals erreicht. Das Symbol `HRTICKS_PER_SEC` ist definiert als die Ticks pro Sekunde. Die Funktion

```
hrtime_t gethrtime(void);
```

liefert die aktuelle Zeit.

FIFO-Handling

Für Linux-Prozesse stehen die Real-Time FIFOs als character devices `/dev/rtf0`, `/dev/rtf1` etc. zur Verfügung. Der Real-Time Prozeß muß die FIFOs explizit erzeugen und ihnen eine Größe geben. Dies geschieht mit der Funktion

```
int rtf_create(unsigned int fifo, int size);
```

Innerhalb der Real-Time Applikation kann mit den Funktionen

```
int rtf_put(unsigned int fifo, char *buf, int count);
int rtf_get(unsigned int fifo, char *buf, int count);
```

auf die FIFOs zugegriffen werden. Eine wichtige Funktion ist der Handler, der immer dann aufgerufen wird, wenn eine Linux-User Programm auf einen FIFO zugreift. Ein solcher Handler wird mit folgender Funktion mit dem FIFO verbunden:

```
int rtf_create_handler(unsigned  int  fifo,
                       int  (* handler)(unsigned int fifo)
);
```

3.1.3 Servos ansteuern mit Real-Time Linux

Servomotoren werden üblicherweise im Modellbau eingesetzt. Handelsübliche Servos haben einen nominalen Winkelbereich von 90 Grad und erzeugen Drehmomente bis zu 150Ncm. Ein Servo verfügt über 3 Anschlußleitungen: Betriebsspannung, Masse und Steuersignal. Alle 20 Millisekunden erwartet der Servo als Steuersignal einen TTL-kompatiblen Impuls von 1 bis 2 Millisekunden Länge. Die Länge des Impulses bestimmt die Stellung des Servos, d.h. 1ms = links, 1.5ms = mitte, 2ms = rechts (vgl. Abbildung 3.1).

Genauigkeit der Servoansteuerung: Um den Servo genau zu positionieren, muß die Länge des Signals exakt eingehalten werden. Eine Abweichung von 10μs entspricht ungefähr einer Abweichung von einem Grad. Messungen an unserer eingesetzten Testmaschine (PII, 333MHz) haben ergeben, daß man mit einer maximalen Latenz von $12\mu s$ rechnen muß. Die durchschnittlichen

[1] Bei der Installation von Real-time Linux wird die Anzahl der Tick pro Sekunde im Betriebssystem verankert. Bei Austausch von Hardware ist somit der Real-Time Kernel neu zu übersetzen.

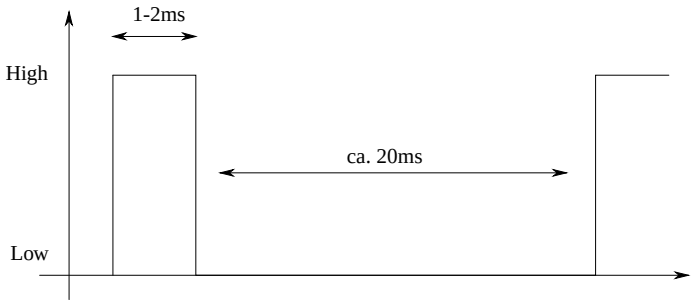


Abbildung 3.1: Steuersignale für den Servo

Auflösung Laserscanner in Punkte pro Scanline	Auflösung Servo in Grad		
	1/3	2/3	1
125	28350	18900	9450
250	56700	37800	18900
500	113400	75600	37800

Tabelle 3.2: Die Auflösung des 3D-Scanners

Abweichung ist $5\mu s$ und daher die Auflösung des Servos 0.5 Grad. Damit ergeben sich – unter variablen Punktauflösungen des Scanners – Auflösungen, die in Tabelle 3.2 dargestellt sind.

Folgende Skizze veranschaulicht den einfachen Anschluß eines Servomotors an die paralleler Schnittstelle.

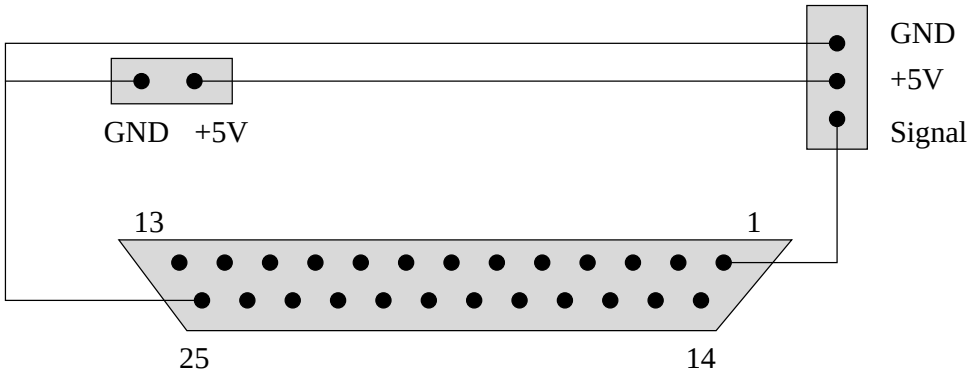


Abbildung 3.2: Anschluß des Servos an die parallele Schnittstelle des Rechners

Das Verfahren zur Servoansteuerung läßt sich auf bis zu 12 Servos pro parallele Schnittstelle einfach erweitern.

Das Real-Time Modul steuert einen Servo an. Der periodische Prozeß in diesem Modul besteht im wesentlichen aus einer Schleife, die endlos durchlaufen wird. Als erstes wird die Zeit in Prozessorticks berechnet und überprüft, ob der Prozeß zu früh ausgeführt wird (*early scheduling*). Wenn

die 20 ms erreicht sind, wird an der Leitung der parallelen Schnittstelle das TTL-Signal angelegt. Danach legt sich der Prozeß eine Zeit lang schlafen. Diese Zeit bestimmt die Stellung des Servos. Anschließend wird das Signal wieder gelöscht und die Kontrolle bis zum nächsten Durchgang an andere Prozesse abgegeben. Damit läßt sich bereits eine Stepper-Funktion realisieren.

Das Modul ist auch in der Lage, den Motor kontinuierlich zu drehen und nach erfolgter Drehung dem Anwendungsprogramm dies mitzuteilen. Zur Kommunikation werden 2 Real-Time FIFOs benutzt.

Durch den ersten FIFO (`/dev/rtf0`) teilt das Anwendungsprogramm über folgendes Telegramm die Anweisungen mit:

```
struct msg_struct{
    char option;
    hrtime_t data1;
    hrtime_t data2;
    hrtime_t time;
};
```

Ist das übermittelte Zeichen ein `N`, so wird der Servo angewiesen, die in `data1` übermittelte Zeit (in Prozessorticks) als Highimpuls auszugeben.
Ist das übermittelte Zeichen `T`, so wird der Servo zwischen den Stellungen, die durch `data1` und `data2` spezifiziert sind, bewegt. Dabei wird jedesmal das auszugebende Signal um `time` Ticks erhöht bzw. erniedrigt. Dadurch konnte auf die Einbindung von floating-point Routinen verzichtet werden. Es ist prinzipiell möglich, floating-point Routinen statisch zu einem Real-Time Modul zu linken, da diese keine UNIX Systemaufrufe enthalten.

Der zweite FIFO (`/dev/rtf1`) wird dazu benutzt, um dem Anwendungsprogramm das Ende einer Drehung mitzuteilen. Es wird nach Beenden der Drehung immer ein `D` in den FIFO geschrieben. Mittels non-blocking Fileoperationen kann das Anwendungsprogramm diesen Status kontinuierlich abfragen.

3.2 Meßdatenverarbeitung

Definition 1 (Scan-Punkt). Ein Scan-Punkt $s_i = (\xi_i, \zeta_i, \phi_i)$ ist ein 3-Tupel, bestehend aus den 2D-Koordinaten $\xi_i, \zeta_i \in \mathbb{N}$ sowie einem Winkel ϕ_i als der Winkel zwischen der aktuellen Scanebene und der (X,Z)-Ebene, der Grundfläche im Raum.

Definition 2 (Scan). Ein Scan $S = (s_i)_{i=1,\dots,n}$ ist eine Folge von Scan-Punkten, welche alle in einer Ebene liegen ($\phi_k = \phi_l \; \forall k,l \in \{1,\dots,n\}$).

Die bisherigen Definitionen beziehen sich auf einen zweidimensionalen Scan, wie er von dem Laserscanner direkt übermittelt wird. Im folgenden nun die Überleitung zum Dreidimensionalen.

Definition 3 (3D-Scan). Ein 3D-Scan $S_{3D} = (S_j)_{j=1,\ldots,m} = (s_{ij})$ besteht aus einer Folge von (ebenen) Scans. s_{kl} bezeichnet somit den k-ten Punkt des l-ten Scans.

Definition 4 (Punkt). Ein gescannter Punkt p wird repräsentiert als ein Tripel $(x,y,z) \in \mathbb{N}^3$. Die Abbildung $(\mathsf{coord}_i)_{i\in\{1,2,3\}} : \mathbb{N}^3 \to \mathbb{N}$ sei definiert als

$$\mathsf{coord}_i\big(p = (x,y,z)\big) := \begin{cases} x & \text{falls } i = 1 \\ y & \text{falls } i = 2 \\ z & \text{falls } i = 3\,. \end{cases} \tag{3.1}$$

Definition 5 (Linie). Eine Linie $l = (p_1, p_2)$ besteht aus Anfangs– sowie Endpunkt, das bedeutet $l \in \mathbb{N}^3 \times \mathbb{N}^3$. Sei l eine Linie, so liefert die Abbildung $\mathsf{head}(l)$ den Startpunkt der Linie, $\mathsf{head}\colon (\mathbb{N}^3 \times \mathbb{N}^3) \to \mathbb{N}^3$. Entsprechend liefert $\mathsf{tail}\colon (\mathbb{N}^3 \times \mathbb{N}^3) \to \mathbb{N}^3$ den Endpunkt von l.

Definition 6 (Fläche). Eine Fläche $f = (l_1, l_2)$ ist definiert als die Fläche, die zwischen zwei Linien, einer Start– sowie einer Endlinie, eingeschlossen wird, $f \in \big((\mathbb{N}^3 \times \mathbb{N}^3) \times (\mathbb{N}^3 \times \mathbb{N}^3)\big)$.

Definition 7 (3D-Scan (2)). Als Alternative zu Definition (3) läßt sich ein 3D-Scan auch betrachten als eine Folge von Punkten oder Linien aus dem 3D-Raum:

$$S'_{3D} = (S_j) = (p_{ij}) \qquad \text{bzw.} \qquad S''_{3D} = (S_j) = (l_{ij}). \tag{3.2}$$

3.2.1 Linienerkenner

Der erste Schritt der Meßdatenverarbeitung ist die Linienerkennung. Sie dient in erster Linie zur Datenreduktion für die nachfolgenden Schritte der Segmentierung und Objekterkennung. Linienerkenner auf 2D Laserscannerdaten werden schon recht erfolgreich in herkömmlichen Anwendungen zur Navigation eingesetzt [Arr96]. Die dort eingesetzten Linienerkenner sind speziell für Laserscanner geeignet.

Wir haben zwei Linienerkenner implementiert. Der erste Algorithmus von Surmann und Liang [Pau98] ist ein einfacher Online-Algorithmus zum Detektieren von Linien mit der Zeitkomplexität $\mathcal{O}(n)$. Der zweite Linienerkenner ist die aus der Bildverarbeitung bekannte Houghtransformation [Hab91]. Die Houghtransformation ist recht gut für unsere Zwecke geeignet, da sie auf einem schwarz/weiß Bild arbeitet. Somit sind die Scan-Punkte die einzigen Bildpunkte. Dadurch ist die Houghtransformation immer noch recht schnell, mit der Zeitkomplexität $\mathcal{O}(c \cdot n), \quad c = \sqrt{(\Delta x)^2 + (\Delta y)^2}$, obwohl die Konstante wegen der großen Scanreichweite recht groß werden kann. Desweiteren liefert die Houghtransformation nicht nur die einzelnen Liniensegmente, sondern ebenfalls eine allgemeine Geradengleichung der Form $y = a \times x + b$. Diese Geradengleichung entspricht einer Ausgleichsgerade, die mehrere unterbrochene Liniensegmente verbindet. Derartige Liniensegmente treten beispielsweise an Wänden auf, die durch Nischen, Vorsprünge oder Türen unterbrochen sind.

Einfacher Linienerkenner

Der hier vorgestellte Linienerkenner wurde von Surmann und Liang speziell für 2D-Laserscanner entwickelt. Er kommt bei Robotern, die lediglich über 2D-Scanner zur Hindernisdedektierung verfügen, zum Einsatz. Der größte Vorteil dieses Verfahrens ist die Geschwindigkeit, denn dieser einfache Linienerkenner ist in der Praxis noch deutlich performanter als die ohnehin schon schnelle Houghtransformation.

Der Linienerkenner funktioniert in zwei Schritten,

1. Datenreduktion
2. Linienerkennung

welche sequentiell abgearbeitet werden.

Die Datenreduktion In diesem Schritt werden die Meßwerte pro 2D-Scan (bis zu 500) reduziert. Dabei werden sukzessive Meßwerte genau dann zu einem Punkt zusammengefaßt, wenn ihr Abstand voneinander einen vorher festgelegten Wert nicht überschreitet. Dieser Wert wurde als `stepDistance` bezeichnet und stellt einen Parameter dar, mit dem die Güte der Datenreduktion beeinflußt werden kann.

Die Abbildung 3.3 zeigt einen 2D-Scan, Abbildung 3.4 die dazugehörigen reduzierte Datenmenge. Im Beispiel wurde mit einer `stepDistance` von 6 gearbeitet, d.h. alle Punkte in einem Radius von 6 cm werden zu einem einzelnen Punkt (ihrem Mittelpunkt) zusammengefaßt. Die 250 Meßwerte konnten damit auf 90 reduziert werden.

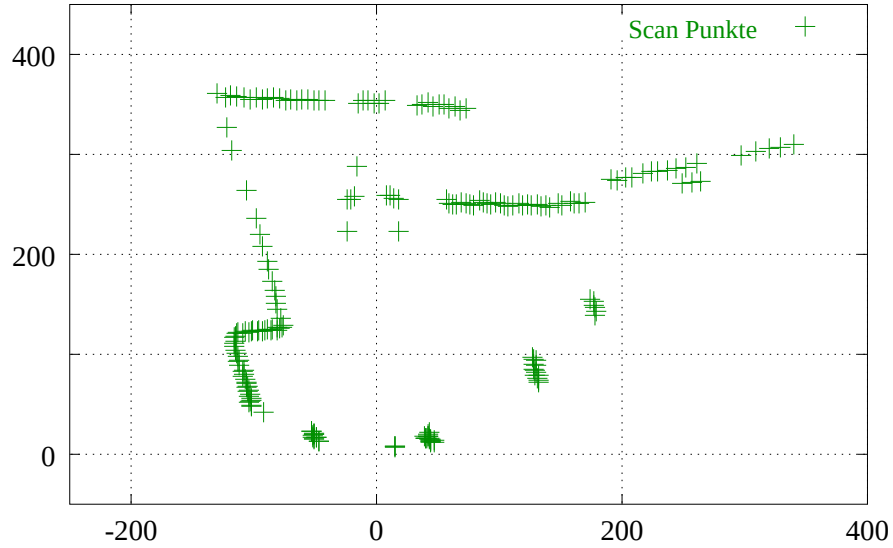


Abbildung 3.3: Beispielscan

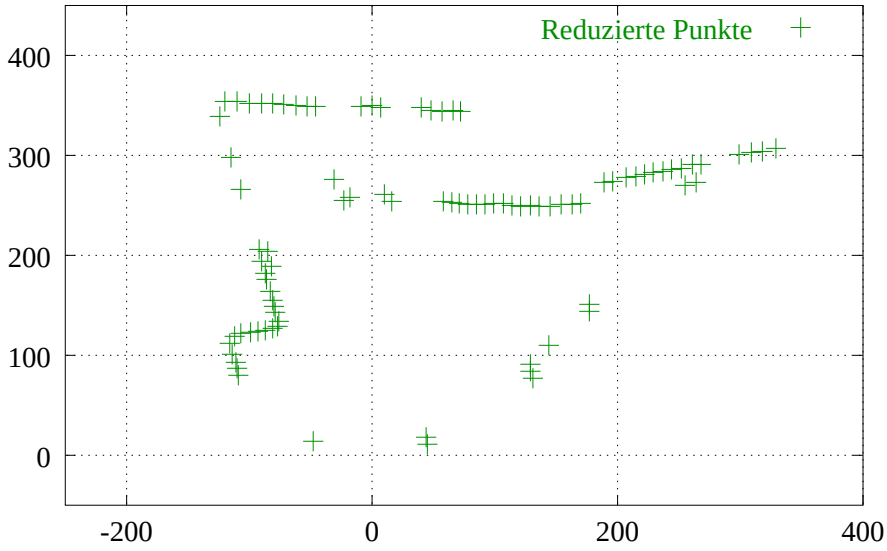


Abbildung 3.4: Illustration der Punkte-Reduzierung

Der einfache Linienerkennungsalgorithmus Der Linienerkenner geht sukzessive alle verbliebenen Punkte durch,um Punkte zu Linien zusammenzufassen. Seien $a_0, a_1, \ldots, a_n$ die Punkte. Weiterhin sei bereits bekannt, daß $a_i, \ldots, a_j$ auf einer Linien liegen. Nun betrachtet der Algorithmus den Punkt a_{j+1}. Es werden folgende Werte berechnet:

$$\texttt{distance} := \left|\left|a_j, a_{j+1}\right|\right|$$

$$\texttt{sum\_distance} := \sum_{t=i}^{j} \left|\left|a_t, a_{t+1}\right|\right|$$

$$\texttt{direct\_distance} := \left|\left|a_i, a_{j+1}\right|\right|$$

$$\texttt{two\_distance} := \left|\left|a_{j-1}, a_j\right|\right| + \left|\left|a_j, a_{j+1}\right|\right|$$

$$\texttt{two\_d\_distance} := \left|\left|a_{j-1}, a_{j+1}\right|\right|$$

$$\texttt{ratio} := \frac{\texttt{direct\_distance}}{\texttt{sum\_distance}}$$

$$\texttt{ratio2} := \frac{\texttt{two\_d\_distance}}{\texttt{two\_distance}}$$

Die Linie $a_i, \ldots a_j$ wird nun um den Punkt a_{j+1} erweitert, wenn *alle* der folgenden Bedingungen erfüllt sind:

1. $\texttt{ratio} > 1 - \frac{0.3}{(1+1.5\cdot(j+1))}$

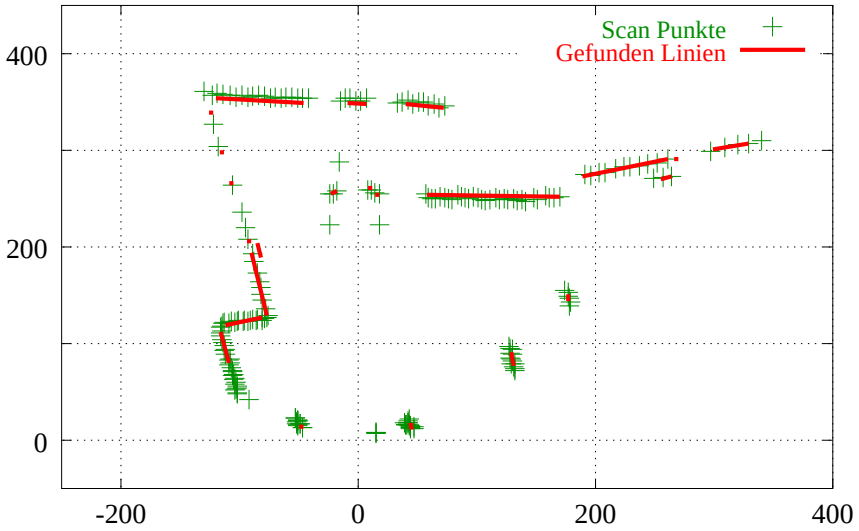


Abbildung 3.5: Die Wirkungsweise des einfachen Linienerkenners

2. `ratio2` > 0.8
3. `distance` $< 3 \cdot$ `stepDistance`.

Wie man leicht sieht, wird mit `ratio` und `ratio2` die Linienerkennung kontrolliert. Die 3. Ungleichung drückt aus, inwieweit Linien ohne vorliegende Meßpunkte extrapoliert werden. Die in den Ungleichungen auftretenden Konstanten wurden von uns empirisch bestimmt.

Abbildung 3.5 zeigt das Resultat des Linienerkenners auf den Meßpunkten aus Abbildung 3.4 (eingezeichnet in die Originaldaten). Ein Vergleich der beiden Abbildungen zeigt, daß alle Punkte, die nach dem Reduzieren übrig geblieben sind, auch Linien zugeordnet werden konnten.

Houghtransformation

Die Houghtransformation dient zum Erkennen von geometrischen zweidimensionalen Objekten [McL98, URLHou]. Dazu wird eine (ρ, θ)–Parametrisierung des Linienraums benutzt, wobei θ der Winkel des Normals zu der Linie ist und ρ die Länge des Normals, also der Abstand des Punktes zum Ursprung des Bildes.

Alle Punkte einer Linie erfüllen durch diese Parametrisierung die Gleichung

$$\rho = x \cdot \cos(\theta) + y \cdot \sin(\theta). \tag{3.3}$$

Ein Beispiel soll das Verfahren demonstrieren: Gegeben seien vier Punkte (vgl. Abbildung 3.7): $(50, 200)$, $(75, 200)$, $(100, 200)$ und $(125, 100)$. Wie man leicht sieht, liegen drei der vier Punkte auf einer Geraden. Diese drei Punkte überlagern sich im (ρ, θ)-Raums konstruktiv und bilden im Histogramm dann ein Maximum.

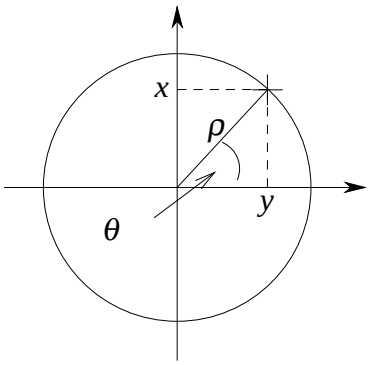


Abbildung 3.6: Der Kreis für die Berechnung der Houghtransformation

Diese Transformation wird für alle Punkte des Scans berechnet und das Ergebnis in einem Histogramm aufsummiert. Dabei wird sowohl der Winkel $\theta(j)$ als auch $\rho(k)$ diskretisiert [URLHou].

```
for (i=0; i < nr_pts; i++) {
   for (j=0; j < resolution; j++) {
      // theta in [-PI/2,PI/2)

      // Die Winkel theta sind in einem Array hinterlegt
      rho = cos_theta[j] * (double) x[i] + sin_theta[j] *
            (double) y[i];
      k   = (int) ((resolution) * (rho / (2.0 * max_rho_d) +
            0.5));

      histogram[j][k]++;
   }
}
```

Danach wird im Histogramm das globalen Maximum bestimmt. Damit errechnet man eine Geradengleichung und markiert alle zu dieser Geraden gehörigen Punkte. Gleichzeitig werden alle Linien, d.h. Geradensegmente als Ausgabe für die Houghtransformation, bestimmt. Viele Parameter bestimmen die Güte der aus den Geraden extrahierten Linien und wurden dahingehend angepaßt, daß möglichst korrekt im *nahen* Sensorbereich erkannt wird. Anschließend werden die Punkte, die zu einer Gerade gehören, von dem Histogramm abgezogen. Dabei wird nicht nur das Maximum des Histogramms entfernt, sondern auch alle Punkte, die zu der geraden beigetragen haben. Damit senkt sich das Gesamtniveau des Histogramms also ab.

Dieser Prozeß wird iteriert, d.h. die Houghtransformation wird erneut gestartet mit den verbliebenen Punkten. Es wird solange iteriert, bis entweder alle Punkte in Geraden aufgegangen sind (hierbei werden einzelne Punkte als Ein-Punkt-Geraden erkannt), oder nach der 100sten Iteration abgebrochen wird.

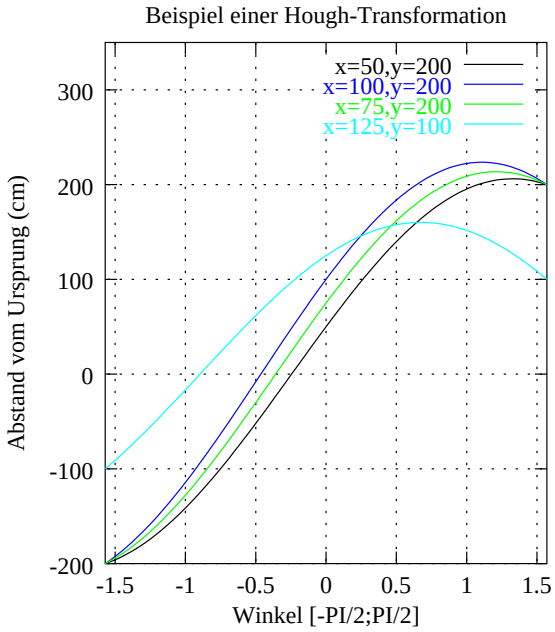


Abbildung 3.7: Houghtransformation mit 4 Punkten

Abbildung 3.8 stellt das zweidimensionale Histogramm dar und veranschaulicht die Entfernung von Geraden.

Bemerkung: Die Houghtransformation liefert Geradengleichungen. Die hier vorgestellte Arbeit benutzt zur Zeit nur die Linien-, d.h. Strecken-Information. Die Houghtransformation jedoch liefert insbesondere Geradengleichungen, welche die Möglichkeit bieten, aufgrund von Hintergrundwissen über die eingescannte Szene verdeckte Linien zu propagieren. Wände lassen sich somit extrapolieren, was für Anwendungen in der Robotik sehr sinnvoll ist.

Zum Abschluß geben wir zwei Beispiele. Abbildung 3.9 zeigt die mittels der Hough-Transformation erkannten Linien zu den in den Abbildungen 3.8 dargestellten Histogrammen. Abbildung 3.10 zeigt den gleichen 2D-Scan wie Abbildung 3.3 und ist hier als direkter Vergleich zu dem einfachen Linienerkenner angegeben.

3.2.2 Meßwertumrechnung

Die beim Einscannen gelieferten Daten – die Punkte eines Scans – müssen nun noch in das kartesische Koordinatensystem $\mathbb{R}^3$ eines 3D-Scans umgerechnet werden. Die Scan-Punkte eines Scans sind gegeben durch ξ, ζ Koordinaten und den aktuellen Neigungswinkel des Scanners, ϕ.

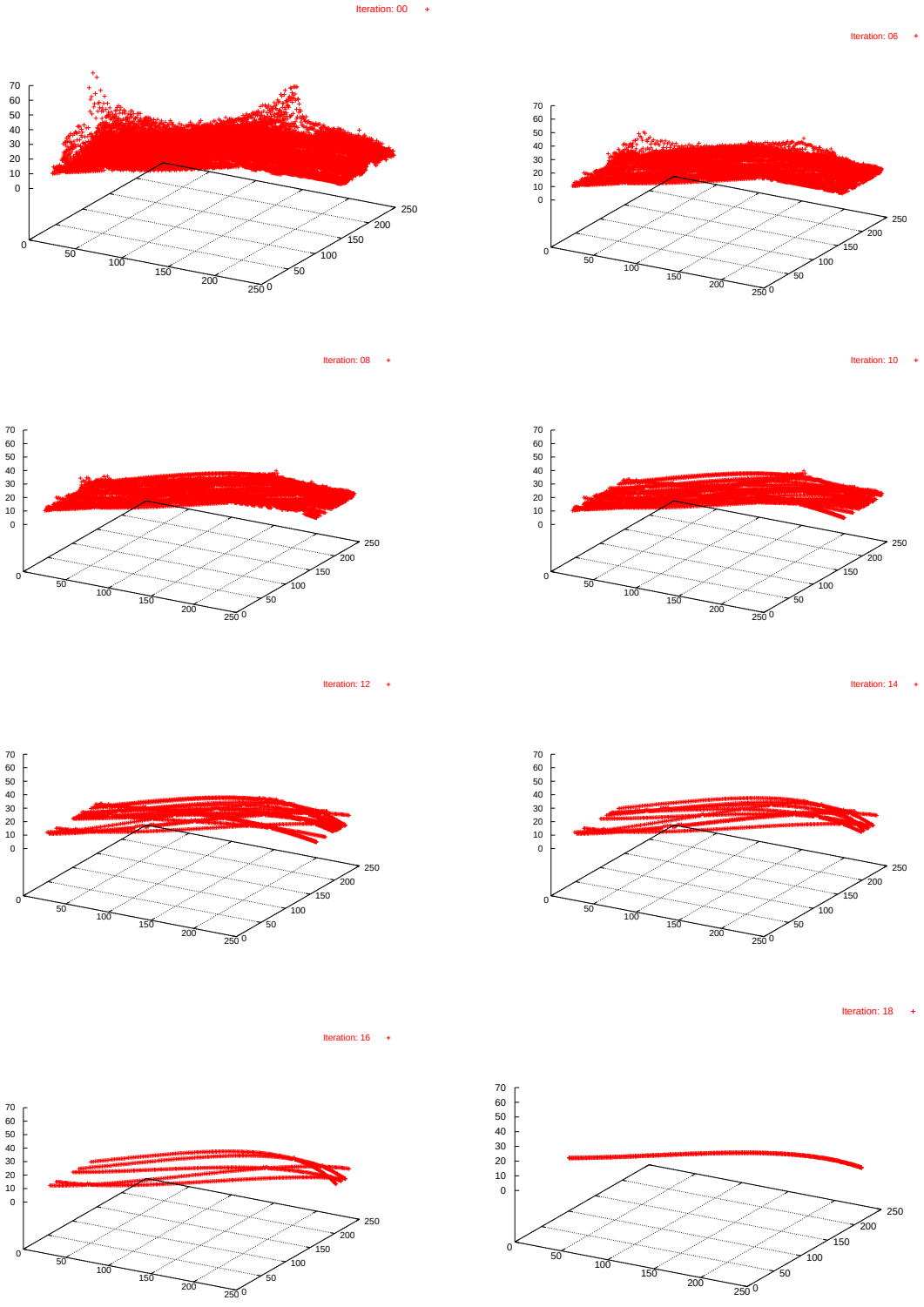


Abbildung 3.8: Histogramme zur Houghtransformation

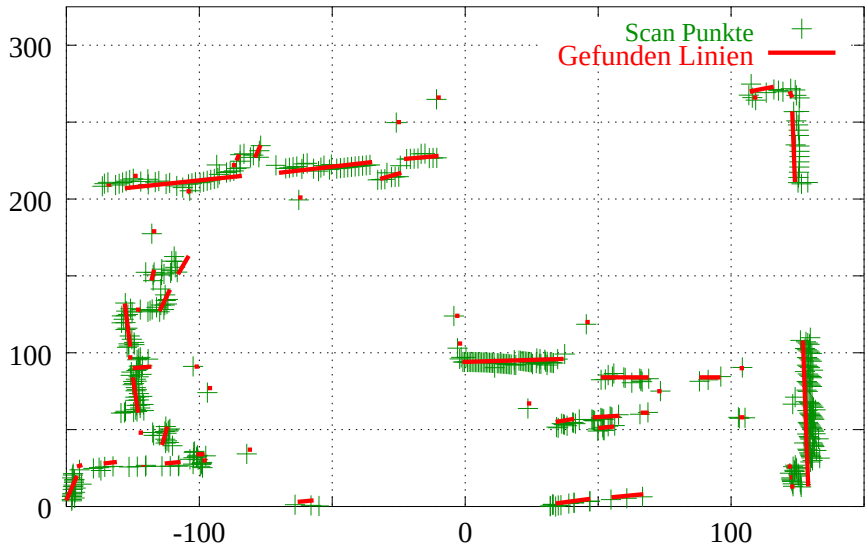


Abbildung 3.9: Erkannte Linien mit Houghtransformation

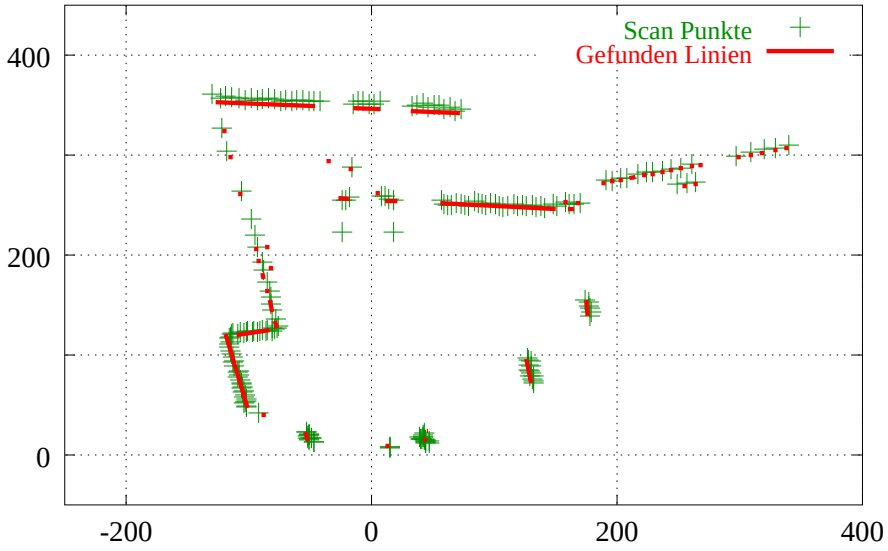


Abbildung 3.10: Erkannte Linien mit Houghtransformation – analog zu Abbildung 3.5

Die (x,y,z)–Koordinaten werden aus den Meßwerten (ξ,ζ) berechnet (vgl. Abbildung 3.11) durch

$$x = \xi \tag{3.4}$$
$$y = \zeta \cdot \sin(\phi), \tag{3.5}$$
$$z = \zeta \cdot \cos(\phi). \tag{3.6}$$

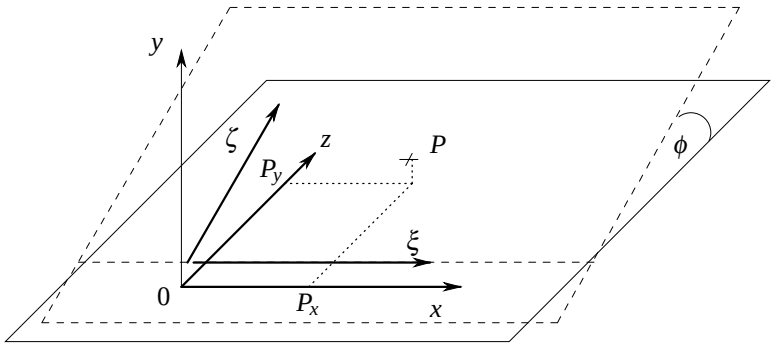


Abbildung 3.11: Umrechnung der Koordinaten

3.2.3 Berechnung der Texturen

Der 3D Scanner ist zusätzlich mit einer Kamera ausgestattet, um Bilder für Texturen aufzunehmen, die dann auf die Objekte abgebildet werden können.

Bei der Texturberechnung wird jedem Punkt eindeutig ein Punkt auf dem Foto, beziehungweise auf dem aus mehreren Fotos zusammengesetzten Bild, zugeordnet. Als Textur-Koordinaten werden im folgenden die zu einem Scan-Punkt gehörigen Koordinaten auf dem Bild bezeichnet.

Um die Textur-Koordinaten eines Meßpunktes zu berechnen, soll jeder Meßpunkt P auf eine virtuelle Ebene E projiziert werden (siehe Abbildung 3.12). Dies geschieht online, also während des Scanvorgangs.

Um dies zu erreichen, werden zuerst die Winkel φ_1 (Winkel zwischen (P_z,P_x) und der x-Achse) und φ_2 (Winkel zwischen (P_y,P_z) und der y-Achse und der z-Achse, verschoben um die Kameraposition) berechnet.

$$\varphi_1 = \frac{\pi}{2} - \arctan\left(\frac{P_z}{P_x}\right) \tag{3.7}$$
$$\varphi_2 = \arctan\left(\frac{P_y - h(\text{Scanner})}{P_z}\right) \tag{3.8}$$

Mit diesen Winkeln wird dann der Punkt auf eine 2 Meter entfernte Ebene projiziert. Die Projektion $V \to P \to P'$ wird in zwei Projektionen zerlegt, nämlich eine, die in der (y,z) Ebene abläuft,

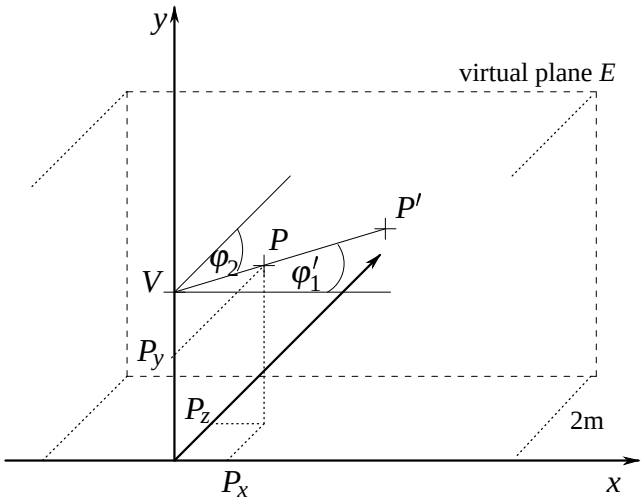


Abbildung 3.12: Projektion eines Meßpunktes auf eine virtuelle Ebene

und eine, die in der (x,y) Ebene abläuft. Das Ergebnis der beiden Projektionen ergibt sich durch Zusammensetzung der Projektionen.

$$x = x + \tan(\varphi_1) \cdot \left(\mathsf{dist}(\text{Plane}) - P_z\right) \tag{3.9}$$

$$y = y + \tan(\varphi_2) \cdot \left(\mathsf{dist}(\text{Plane}) - P_z\right) \tag{3.10}$$

Anschließend wird das zweidimensionale Abbild des Scans noch so skaliert, daß Punkte, von denen Texturinformationen vorliegen, x Werte zwischen 0 und 506/1024 und y Werte zwischen 0 und 997/2048 besitzen. Dabei ist 506×997 die Auflösung des aus mehereren Fotos bestehenden Bildes und 1024×2048 die Auflösung der Textur innerhalb des OpenGL Programms (vgl. Abschnitt 6.5.2)

Bei dem verwendeten Aufbau exisistiert nicht von allen Punkten eine Texturinformation, da der Öffnungswinkel der Kamera wesentlich kleiner als 180 Grad ist.

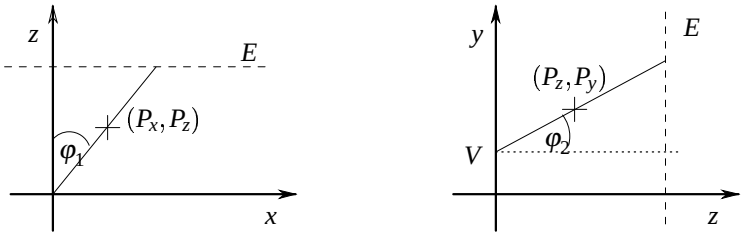


Abbildung 3.13: Darstellung der zu berechnenden Winkel

3.2.4 Flächen-Segmentierung

Die Flächensegmentierung beruht auf der Annahme, daß eine eingescannte Fläche (z.B. eine Wand) in mehreren — in etwa aufeinander folgenden — Scans durch hinreichend nahe übereinander liegende Linien approximiert wird.

Gegeben sei ein Objekt, welches im Sichtbarkeitsbereich des Scanners liegt. Die erste (unterste) Linie, die zu diesem Objekt gehört, sei zuerst im k-ten Scan S_k sichtbar und werde mit l_{ik} bezeichnet. Wenn in dem darauffolgenden Scan S_{k+1} die Prädiktion einer potentiellen Fläche validiert wird, d.h. eine Linie $l_{j,k+1}$ existiert, so daß die Endpunkte beider Linien hinreichend genau übereinstimmen, wird die Anfangslinie aus Scan k zu einer Fläche expandiert.

Dieser Vergleich der Endpunkt ist in Abbildung 3.14 zu sehen. Jedoch beschränkt sich der Test nicht nur auf die reine euklidische Distanz der Endpunkte, sondern schließt auch die Überprüfung des Winkels $\measuredangle(l_{i,k}, l_{j,k+1})$ zwischen beiden Linien mit ein, der eine Schranke ε_2 nicht überschreiten darf. Ansonsten besteht insbesondere bei kürzeren Linien die Möglichkeit, daß sie selbst dann gematcht werden, wenn ihre Orientierungen stark voneinander differieren.

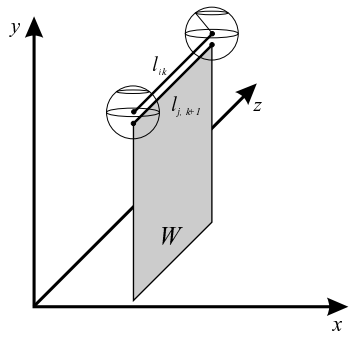


Abbildung 3.14: Expansion einer Fläche durch eine weitere gematchte Linien

Die eigentliche Segmentierung erfolgt in drei Schritten: Gegeben sei ein Scan $S_j = (l_{ij})_{i=1,\ldots,n}$, bestimmt durch die darin erkannten Linien. Jede dieser neuen Linien wird nun wie folgt unterschieden:

1. Stellt l_{ij} eine Expansion einer bereits gefundenen Fläche dar?
   Dazu wird die Datenstruktur `surface_queue`[2] durchlaufen auf der Suche nach einer Fläche, deren obere Linie mit l_{ij} gematcht werden kann (d.h. diese beiden Linien müssen oben beschriebene Kriterien erfüllen). Wenn eine solche Fläche existiert, wird diese um l_{ij} erweitert.
2. Andernfalls wird in der Struktur `lines_queue` – die alle (zumindest bis zu diesem Zeitunkt noch) singulären Linien verwaltet – nach einer Linie gesucht, welche mit l_{ij} gematcht werden kann und höchstens 3 Scans zuvor aufgenommen worden ist. Auf diese Weise ist

[2] vergleiche Abschnitt 5, insbesondere Abbildung 5.2.

das Verfahren hinreichend fehlertolerant bei Scans, in denen die Linien (lokal) nur unzureichend erkannt werden konnten.
Diese beiden Linien bilden nun eine neue Fläche ($\rightsquigarrow$ Verschiebung in `surface_queue`), die mit hoher Wahrscheinlichkeit innerhalb der nächsten Scans weiter expandiert werden wird.

3. Im dritten Fall wird l_{ij} als singuläre Linie in `lines_queue` aufgenommen, um bei der Bearbeitung eines späteren Scans möglicherweise in Schritt (2) als Grundlinie einer Fläche erkannt zu werden.

Am Ende des Algorithmus sind nun jene Linien, die hinreichend gut eine Fläche in der eingescannten Szene approximieren, nun auch intern zu einer Fläche zusammengefügt. Die in der Szene erkannten Flächen dienen nicht nur der Visualisierung (z.B. durch Einfärbung mittels einer aus dem Kamerabild extrahierten Textur) und der Objektsegmentierung. Sie stellen auch eine „Verbindung“ zwischen verschiedenen Scans dar: Marken, die es ermöglichen, mehrere sukzessive aufgenommenen Scans zu einer großen Aufnahme zu vereinen.

Kapitel 4

Offline-Algorithmen zur Objektsegmentierung

Im folgenden werden Strategien zur Interpretation der Scan-Ergebnisse erläutert, insbesondere zur Segmentierung der Bildelemente in einzelne, größere Objekte. Im Gegensatz zu dem vorherigen Kapitel sind dies Algorithmen, die aufgrund ihrer Komplexität offline durchzuführen sind.

4.1 Projektion der Daten

Der Scanner liefert die Daten der Umgebung in einzelnen (X,Z)–Schnitten durch den Raum. Ein Scan läßt sich also als eine Art „Draufsicht“ auf den Raum von oben aus sehen, mit steigender Scannummer steigt insbesondere die Höhe des Schnittes.

Da diese Ansicht jedoch ausgesprochen unintuitiv ist, bestand der erste Versuch nun darin, die Daten eines kompletten 3D-Scan in (X,Y)–Richtung zu projizieren, um auf diese Weise eine Art Frontalansicht zu erhalten. Dies bedeutet, daß – vom Betrachter aus gesehen – mit wachsender Scannummer die (X,Y)–Ebene weiter nach hinten wandert.[1]

Es folgen nun – sofern nicht anders erwähnt – zunächst einmal Algorithmen, die auf diesen *2D-Projektionen* arbeiten. Später jedoch wird ein Wechsel zu Verfahren stattfinden, die direkt in dem 3D-Bild arbeiten.

4.1.1 Modifizierung der Daten

Zur besseren Handhabung der quasi-kontinuierlichen Daten des Scanners wurde zunächst der Datenstrom nach Abtastung in X–Richtung und Quantisierung in Y–Richtung in einer Booleschen Matrix $\Phi \in \mathbb{B}^{n \times m}$ abgespeichert, also doppelt gleichförmig digitalisiert (Abbildung 4.2). In diesem

[1] Ein animiertes GIF-Bild ist unter [URLGif] abrufbar.

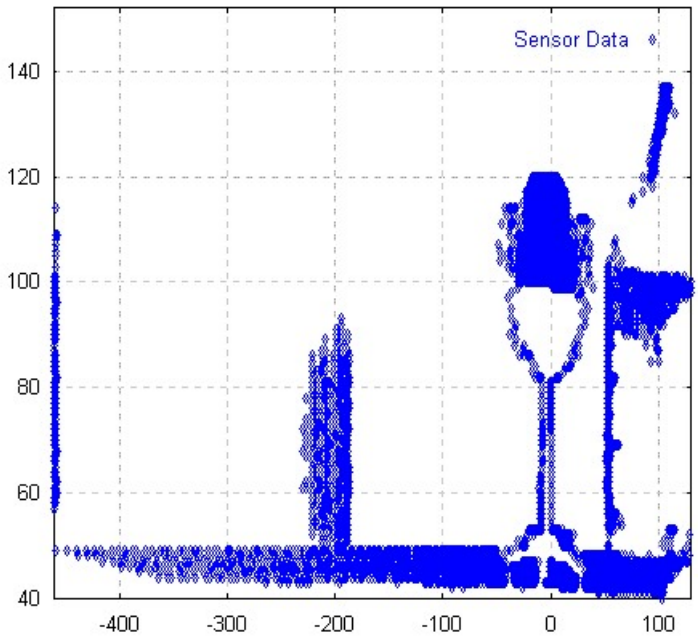


Abbildung 4.1: Projektion eines 3D-Scans auf die (X,Y)–Ebene

digitalisierten Bild galt es nun, die *Kanten* der Objekte bzw. Objektflächen zu extrahieren: mittels Ortsraumfilterung sollten augefüllte Flächen eliminiert und einzig Pixel, die eine Hintergrund- von einer Objektfläche trennen, übriggelassen werden.

Der naheliegendste Ansatz bestand in der Anwendung eines *Gradientenfilters* [Hab91, Shi87] auf das Bild: Jeder Bildpunkt $\Phi(\xi,\zeta)_{\substack{\xi\in 2,\ldots,n-1\\ \zeta\in 2,\ldots,m-1}}$ mit Grauwert $\pi\big(\Phi(\xi,\zeta)\big)$ bekommt den neuen Grauwert

$$\pi'\big(\Phi(\xi,\zeta)\big) := \sum_{i=-\rho}^{\rho}\sum_{j=-\rho}^{\rho} \pi\big(\Phi(\xi,\zeta)\big)\cdot\Psi(i,j) \tag{4.1}$$

zugewiesen. Der Einfachheit halber wird im folgenden der Radius $\rho = 1$ gesetzt.

Dabei ist die Filter-Matrix Ψ je nach zu verwendetem Filtern (X–Gradient, Y–Gradient) zu wählen als:

$$\underbrace{\begin{bmatrix} -1 & -1 & -1\\ 0 & 0 & 0\\ 1 & 1 & 1 \end{bmatrix}}_{(X\text{–Gradient})} \quad \text{oder} \quad \underbrace{\begin{bmatrix} -1 & 0 & 1\\ -1 & 0 & 1\\ -1 & 0 & 1 \end{bmatrix}}_{(Y\text{–Gradient})}$$

Es werden bei Verwendung der X–Gradientenmatrix jedoch lediglich *horizontale* Kanten in dem Bild erkannt, bzw. nur *vertikale* Kanten bei der zweiten Matrix. Abhilfe schafft das Kombinieren beider Filtermasken, jedoch brachte dieser Versuch auf den Testdaten noch keinen überzeugenden Erfolg.

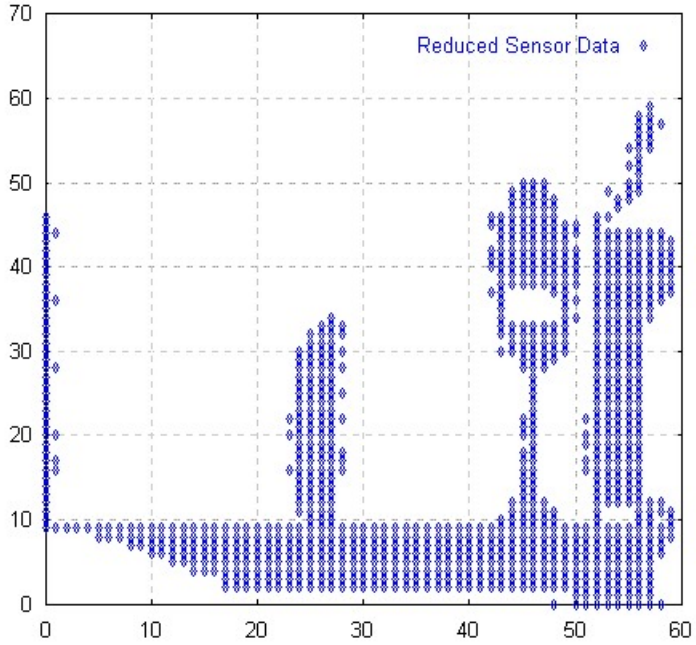


Abbildung 4.2: Digitalisierung der Daten

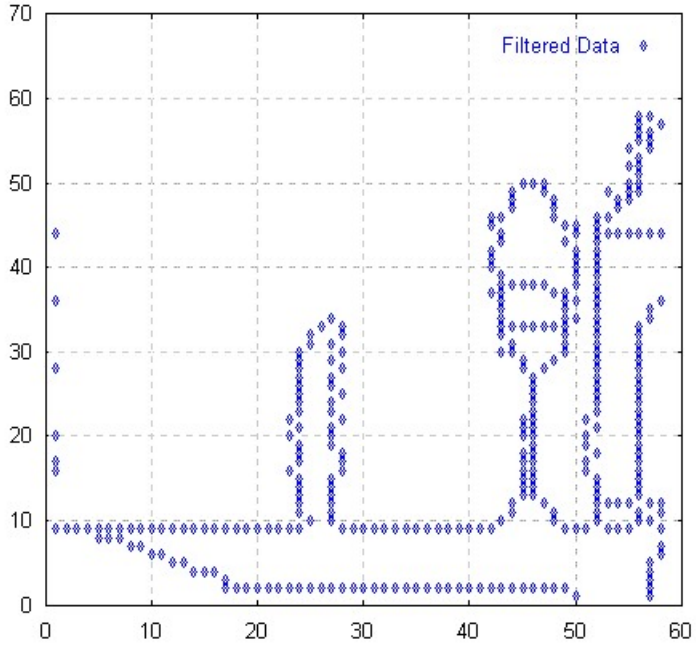


Abbildung 4.3: Anwendung eines Laplacefilters auf das digitalisierte Bild

Ein sehr viel besseres Ergebnis lieferte ein *Laplace-Filter* mit folgender Filtermaske ($\rho = 1$):

$$\Phi := \begin{bmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{bmatrix}$$

Dieser Filter erkennt auch nichtlineare Krümmungen im Grauwertverlauf des Bildes unabhängig von ihrer Richtung (siehe Abbildung 4.3). Da er im allgemeinen auf Rauschen empfindlich reagiert, wird gewöhnlich eine Glättung des Bildes vorgeschaltet, was jedoch bei Tests auf unseren Daten keinen positiven Erfolg brachte. Das mag daran liegen, daß im allgemeinen Fall von *Grauwertbildern* ausgegangen wird, hier jedoch *binäre* Bilddaten vorliegen.

4.1.2 Vektorisierung

Der vorherige Schritt liefert ein Bild, welches lediglich die relevanten Kanten der gescannten Objekte beinhaltet. Jedoch handelt es sich hier immer noch um Bitmap-Bilder, ist für die nachfolgende Flächen– und Objektsegmentierung in dieser Form also ungeeignet.

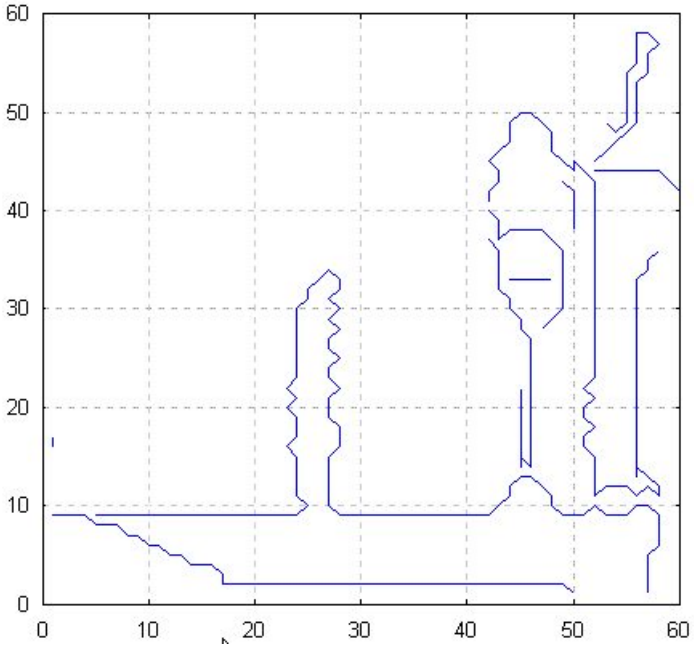


Abbildung 4.4: Ergebnis der Linienerkennung

Mit Hilfe des im Abschnitt 3.2.1 beschriebenen Linienerkennungs–Algorithmus wird nun das Kantenbild vektorisiert, d.h. es wird in eine Anzahl von Linien transformiert. Diese Linien, bestehend aus Anfangs- und Endpunkt, ermöglichen nun die weitere Untersuchung des Scans wie die Segmentierung und anschließende Erkennung von Objekten.

Ein mögliches Ergebnis der Vektorisierung ist in Abbildung 4.4 zu betrachten. Jedoch arbeitet der verwendete Linienerkennungs–Algorithmus naturgemäß unter Verwendung einer Vielzahl von heuristischen Parametern, so daß an dieser Stelle eine Optimierung dieser Parameter aufgrund einer größeren Anzahl von Bildvorlagen notwendig ist. Dabei ist die „optimale“ Einstellung allerdings nicht eindeutig, je nach Zielsetzung der Vektorisierung kann eine Änderung einiger Einstellungen vonnöten sein. So ist es hier beispielsweise sinnvoll, von einer allzu perfekten Nachbildung des Ausgangsbildes abzusehen (es ist durchaus möglich, jede Ecke und Kante durch eine eigene Linie beschreiben zu lassen), da solch ein Ergebnis zwar optisch sehr realistisch aussieht, jedoch beispielsweise eine sinnvolle Flächenerkennung als nahezu unmöglich gestaltet.

4.1.3 Evaluation

Dieser erste Versuch der Aufbereitung eines 3D-Scans erschien recht naheliegend, ist dann aber zunächst einmal zugunsten des im folgenden beschriebenen Vorgehens nicht weiter verfolgt worden [Blo82]. Grund dafür waren theoretische Überlegungen, daß mit dieser Projektion der dreidimensionalen Daten auf eine 2D-Ebene ein Informationsverlust einhergeht: dem Programm liegt ein $2\frac{1}{2}$–dimensionales Abbild eines Objektes (beispielsweise eines Stuhles) vor, jedoch werden nur 2 Dimensionen wirklich benutzt. Somit gingen die nächsten Versuche – einschließlich der aktuellen Implementation – in die Richtung, komplett auf den 3D-Daten zu arbeiten. Jedoch wurde der Algorithmus zur *Vektorisierung* beibehalten, da dieser Schritt eine notwendige Grundlage für eine Szeneninterpretation sein dürfte.

Es sei jedoch angemerkt, daß dieses Vorgehen trotz der damit einhergehenden Informationsreduzierung durchaus seine Berechtigung hat. Schließlich soll es nicht zur kompletten Interpretation der Szene dienen, sondern lediglich das *Erkennen* eines Objektes erleichtern, *nachdem* die Szene in einzelne Objekte/Cluster segmentiert worden ist. Eingedenk der bisher noch schwer zu bewertenden Ergebnisse der im folgenden skizzierte Algorithmen ist es durchaus möglich, daß der Ansatz der Koordinatenprojektion wieder als additives Vorgehen Verwendung findet, um die bereits implementierten Algorithmen bei der Aufgabe der Objekterkennung zu unterstützen.

Von dem ersten Versuch weitgehend abstrahiert, besteht die aktuelle Umsetzung schematisch gesehen aus folgenden Schritten:

Linien-Erkennung: Wie bereits beschrieben liefert der Scanner die Daten in diskreten Schritten, die jeweils einen Scan der (ξ, ζ)–Ebene darstellen. Sobald solch ein Scan abgeschlossen ist, wird der Linienerkenner (Abschnitt 3.2.1) auf dieser Ebene gestartet. Die resultierenden Linien (also das vektorisierte Bild) werden gespeichert und weiterverarbeitet, indem sie in Beziehung zu vorhergehenden bzw. nachfolgenden Scans gebracht werden.

Flächen-Segmentierung: In diesem Schritt wird versucht, Linien aus unterschiedlichen Scans zu Flächen zusammenzufügen (Abschnitt 3.2.4). So wird eine eingescannte Wand beispielsweise nach dem ersten Schritt durch eine vertikal verlaufende Ansammlung von nahezu

parallelen Linien repräsentiert. Die Flächensegmentierung soll nun alle diese Linien zu einer großen Fläche zusammenfügen.

Objekt-Segmentierung: Schließlich werden singuläre (also nicht zu Flächen segmentierte) Linien sowie kleinere Flächen, die selbst nur einen Teil eines Objektes darstellen, zu größeren Objekten zusammengefügt.

Dieses Vorgehen wird im folgenden näher beschrieben.

Nähere Informationen über den genauen Ablauf und implementationsspezifische Details sind in Abschnitt 5, Seite 55 zu finden.

4.2 Objekt-Segmentierung

Alle bisherigen Elemente (Punkte, Linien, Flächen) sind im Grunde genommen lediglich geschickte „Umstrukturierungen“ der 3D-Daten gewesen. Mit der Datenstruktur `object` hingegen wird eine neue Klasse eingeführt, die ermöglichen soll, die einzelnen Objekte der Szene effizient zu segmentieren und später einmal auch zu klassifizieren. Aus diesem Grund ist `object` auch die einzige Element-Klasse, die nicht – direkt oder indirekt – Punkte referenziert, sondern eigenständig neue Informationen einfügt — die sinnvollerweise jedoch in Termen der schon bekannten Elemente implementiert worden sind.

Ziel eines solchen Objektes ist es nun, hinreichend nahe beieinander liegende Elemente – dies sind primär kleinere Flächen und Anhäufungen von singulären Linien, jedoch können auch Punkte/Punktwolken durchaus von Interesse sein – zu clustern. Diese Cluster werden mit einer *Bounding Box* umgeben und im folgenden als ein einzelnes Objekt betrachtet; jedoch bleiben Referenzen auf diese das Objekt bestimmenden Elemente weiterhin gespeichert, um spätere direkte Zugriffe zu ermöglichen.

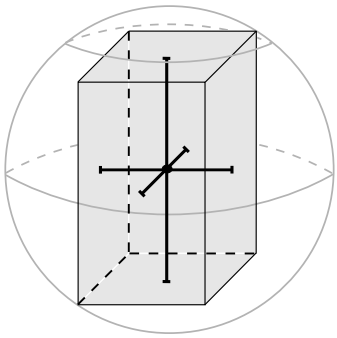


Abbildung 4.5: Repräsentation eines Objektes

Der Objektsegmentierung liegt die Idee zugrunde, daß die oben beschriebenen, noch verbleibenden Elemente sich im allgemeinen zu sinnvoll segmentierbaren Wolken gruppieren, die ein Clu-

stering ermöglicht. Sei beispielsweise ein Stuhl gegeben. Dieser wird im 3D-Scan zwar ein recht klares Bild aus Einzel-Punkten ergeben, jedoch schon die Linienerkennung wird bestenfalls viele kurze, gegeneinander verdrehte Linien finden, welche unmöglich zu einer (großen) Fläche segmentiert werden können (was im Übrigen trivialerweise auch nicht erwünscht wäre).

Wenn jedoch nun ein Element dieses Stuhles, beispielsweise eine kleine Fläche, die die Rückenlehne approximiert, oder auch nur eine einzelne Linie, als Ausgangspunkt genommen wird und davon ausgehend alle Elemente gruppiert werden, welche sich in hinreichend enger *Nachbarschaft* zu diesem Ausgangspunkt befinden, so werden im Idealfall auf diese Weise alle Elemente, die den Stuhl bestimmen, zu einem Objekt geclustert.

Zum besseren Verständnis sei noch erwähnt, daß bei oben beschriebenem Szenario der „Ausgangspunkt“ selber schon als Objekt betrachtet wird, welches es im folgenden gilt zu vergrößern, zu expandieren. Durch das Aufnehmen weiterer Elemente, die sich hinreichend nahe befinden, „wächst“ das Objekt, es vergrößert sich in seinen Ausmaßen. Auf diesen Sachverhalt wird jedoch in Abschnitt 4.2.1 noch näher eingegangen.

4.2.1 Algorithmus

Die eigentliche Objektsegmentierung verläuft in folgenden Schritten:

(0) In einem Vorverarbeitungs-Schritt werden alle Flächen, deren Fläche einen Schwellwert übersteigt, direkt als Objekte klassifiziert. Dies werden im allgemeinen Wände, Schränke und ähnliche große, flächige Gegenstände sein.

(1) Die erste Fläche wird aus der Menge der gespeicherten Flächen entfernt und in ein Objekt transformiert:
Es muß der Mittelpunkt der Fläche bestimmt werden sowie die Achsen so gesetzt, daß der daraus aufgespannte Würfel die gesamte Fläche umschließt. Ferner ist der Radius so zu wählen, daß der Würfel seinerseits von der Kugel vollständig umschlossen wird.

(2) Nun werden die drei Elemente-Listen, die ja stets alle bisher noch singulären Elemente beinhalten, durchlaufen auf der Suche nach Elementen, welche nahe genug zu dem neuen Objekt liegen. Bildlich gesehen werden also alle Elemente in der Nähe des Objektes „aufgesogen“, welches sich dadurch immer weiter aufbläht, bis es (im idealen Fall) alle an dieser Stelle zusammenliegenden Elemente geclustert hat.

Je nach Einstellung zugehöriger heuristischer Parameter werden dabei offensichtlich nahe beieinander stehende Objekte zu einem einzigen verschmolzen. Dies ist zur Erkennung von Hindernissen, denen ein Roboter auszuweichen hat, gerade erwünscht. (Für dicht möblierte Räume kann sogar nur eine einzige Bounding Box existieren.)

(3) Die letzten beiden Schritte werden mit allen anderen Einträgen der Elemente-Listen wiederholt.

Resultat: Der Algorithmus liefert eine Aufteilung der Scene in Objekte dergestalt, daß Akkumulationen von Punkten, Linien und/oder Flächen zu einzelnen Objekten zusammengefaßt und dabei von einer Bounding Box umschlossen werden. Diese Bounding Box stellt nicht nur einen wichtigen Schritt zur Objekterkennung dar. Vielmehr ermöglicht sie ferner die effiziente Navigation um dreidimensionale Hindernisse herum, deren exakte Form zu diesem Zweck nicht notwendig ist [Sur01].

Kapitel 5

Programminterna

Dieses Kapitel stellt den internen Ablauf der Scanner-Applikation näher vor: die parallele Architektur, die Klassenhierarchie sowie das JAVA-Interface zur benutzerfreundlichen Bedienung des Programmes.

5.1 Ablauf-Skizze

Das Programm verläuft in folgenden Schritten:

Zu Beginn werden für den weiteren Programmablauf relevante Einstellungen aus der Konfigurationsdatei `scanner.cfg` gelesen. Diese Parameter lassen sich grob in zwei Gruppen unterteilen:

- Zum einen sind dies rein Hardware-bedingte Größen, die durch den Aufbau des 3D-Scanners gegeben sind oder auf diese direkte Auswirkung haben, wie z.B. die Höhe des Gerätes, der Start– sowie End–Drehwinkel, die Schrittweite, die maximal sichtbaren Koordinaten der Kamera, etc.
- Zum anderen sind dort Variablen einstellbar, die den Ablauf des Programmes beeinflussen. So ist es an dieser Stelle möglich, eine Reihe von heuristischen Parametern zu wählen, wie beispielsweise die in den Formeln (5.1) – (5.3) benutzen Schranken $\varepsilon_1, \varepsilon_2$, die das Matching zwischen zwei Linien bestimmen; weiterhin diverse Variablen zur Steuerung der Linienerkennung sowie Boolesche Parameter, die u.a. die Online-Anzeige während des Scannens ein- oder ausschalten.

Die nächsten Schritte dienen der Vorbereitung der *Parallelisierung* des Programmes (siehe dazu auch Abbildung 5.1): es werden zwei *Pipes* geöffnet, die die Kommunikation zwischen den parallelen Prozessen gewährleisten. Über die erste Pipe werden während eines 3D-Scans die laufenden Daten geschickt, so daß die Scannpunkte online angezeigt werden können. Der Benutzer kann somit den Scan online auf dem Bildschirm verfolgen. Das Anzeigeprogramm `2show` (näher beschrieben in Abschnitt 6), welches diese Daten im Empfang nimmt, wird daraufhin mittels *fork*

als paralleler Prozeß gestartet.

Die zweite Pipe dient der Kommunikation zwischen dem *Hauptprozeß*, der die Daten verarbeitet (siehe unten), und dem *Kindprozeß*, der die Daten liefert. Haupt– und Kindprozeß werden daraufhin gestartet, sobald die Pipe bereitsteht. Per Kommandozeilenparameter kann dem Programm mitgeteilt werden, aus welcher Quelle die Daten stammen sollen (intern wird dazu lediglich ein anderer **Kindprozeß** geforkt):

1. Bei der neuen Aufnahme eines 3D–Scans ist es Aufgabe des entsprechenden Kindprozesses, die Daten, die der Scanner liefert, von der seriellen Schnittstelle abzugreifen und sie an den Hauptprozeß weiterzuleiten sowie gleichzeitig in eine Abfolge von Dateien abzuspeichern. Damit es es nachfolgend möglich, einen 3D–Scan zu simulieren, also auch ohne laufenden Scanner durchzuführen.

   Ferner muß der Motor gesteuert werden, der den Scanner um seine horizontale Achse dreht, und es muß die Kamera ausgelöst werden, deren Bilder später als Quellen der Texturen in dem Anzeigeprogramm dienen. Schließlich hat der Prozeß dafür Sorge zu tragen, „Ausrutscher“ zu eliminieren, indem er bei einem Scan die Anzahl der Punkte, die eindeutig als Meßfehler klassifizierbar sind, mitzählt, korrigiert bzw., falls dies nicht mehr möglich ist, diesen Scan wiederholt.

2. Die zweite Möglichkeit des Aufrufes benutzt die bei einem früheren 3D–Scan gespeicherten Daten-Dateien, d.h. dieser Kindprozeß muß lediglich die Dateien einlesen und sie an den Hauptprozeß senden.

Der **Hauptprozeß** übernimmt die Verarbeitung der Daten. Dies sind im Einzelnen:

1. Die von dem Kindprozeß ankommenden Daten werden im ersten Schritt scanweise *online* verarbeitet:

   - Die Daten eines (ξ, ζ)–Scans werden aus der Pipe gelesen:
     Dies sind die aktuelle Nummer des Scans (nicht notwendigerweise streng monoton steigend), der aktuelle Winkel des Scans sowie Datenpaare $(\xi_i, \zeta_i)_{i=1,\dots,n}$ – die Scanpunkte (n konstant). Letztere werden sofort in einem festen zweidimensionalen Array vom Typ `Point`-Pointer gespeichert, da auf die Originaldaten von nahezu allen anderen Datenstrukturen aus referenziert wird und sie während des gesamten Programmablaufes bestehen bleiben.
   - In dem Scan werden Linien erkannt (siehe Abschnitt 3.2.1).
   - Singuläre Punkte, also Punkte, die nicht auf einer erkannten Linie liegen, werden zur späteren Segmentierung in der Struktur `points_queue` gespeichert.
   - Da die Flächensegmentierung inkrementell arbeitet, wird auch dieser Schritt schon direkt während des Aufnehmens des Scans durchgeführt.

     Das bereits im Abschnitt 3.2.4 angesprochene Kriterium zum Matchen von zwei Linien ist wie folgt realisiert:

Seien l_1, l_2 die zwei zu untersuchenden Linien.Wenn nun gilt:

$$\varepsilon_1 \geq \sqrt{\sum_{i=1}^{3} \left|\mathsf{coord}_i\left(\mathsf{head}(l_1)\right) - \mathsf{coord}_i\left(\mathsf{head}(l_2)\right)\right|^2} \quad \wedge \tag{5.1}$$

$$\varepsilon_1 \geq \sqrt{\sum_{i=1}^{3} \left|\mathsf{coord}_i\left(\mathsf{tail}(l_1)\right) - \mathsf{coord}_i\left(\mathsf{tail}(l_2)\right)\right|^2} \quad \wedge \tag{5.2}$$

$$\varepsilon_2 \geq \arccos \left| \frac{\left(\mathsf{head}(l_1) - \mathsf{tail}(l_1)\right) \cdot \left(\mathsf{head}(l_2) - \mathsf{tail}(l_2)\right)^T}{\left|\mathsf{head}(l_1) - \mathsf{tail}(l_1)\right| \cdot \left|\mathsf{head}(l_2) - \mathsf{tail}(l_2)\right|} \right|, \tag{5.3}$$

so werden l_1 und l_2 als hinreichend *ähnlich* angenommen. Eine Fläche, die beispielsweise l_1 als obere Grenzlinie besitzt, kann nun bis zu l_2 erweitert werden.

- Singuläre Linien – also Linien, welche nicht auf erkannten Flächen liegen – werden entsprechend den Punkten in der Liste `lines_queue` abgespeichert.
- Die Koordinaten der Punkte werden korrigiert (siehe Abschnitt 3.2.2)
- Die korrigierten Punkte sowie ihre Texturkoordinaten werden durch die Pipe an den Anzeigeprozeß gesendet. Weiterhin werden diese Daten in einer Datei abgespeichert, was dem Benutzer die Möglichkeit gibt, das Anzeigeprogramm `2show` zu einem späteren Zeitpunkt als eigenständige Applikation laufen zu lassen.

2. Die nachfolgenden Aufgaben müssen *offline* durchgeführt werden, da im folgenden *alle* Datensätze benötigt werden:
   - Dies ist in erster Linie die Objektsegmentierung.
     Die aktuelle *Repräsentation* eines Objektes (vergleiche dazu auch Abbildung 4.5) besteht im einzelnen aus:
     - Einem Punkt p, der den *Mittelpunkt* des Objektes bezeichnet.
     - Von p ausgehend existieren 3 orthogonale Linien (parallel zur X–, Y– bzw. Z–Achse des Koordinatensystems), die somit einen eindeutigen Würfel aufspannen. Dieser Würfel stellt die oben beschriebene Bounding Box dar, in der alle Elemente, die das Objekt bestimmen, enthalten sind. Wird also ein neues Element in das Objekt mit aufgenommen, wird sich im allgemeinen dieser Würfel weiter ausdehnen.
     - Ferner führt jedes Objekt noch einen Radius r mit sich: Das Paar (p, r) beschreibt eine Kugel, welche das gesamte Objekt (inklusive Bounding Box) einfaßt. Auf diese Weise ist es möglich, einen sehr *schnellen* Test durchzuführen, ob überhaupt eine Möglichkeit besteht, daß das zu untersuchende Element sich nahe genug an der Bounding Box des Objektes befindet, um aufgenommen zu werden.
       Dies bedeutet, daß das Enthaltensein des Elementes in der umschließende Kugel natürlich keineswegs eine hinreichende Bedingung ist, um als Teil des Objektes

betrachtet werden zu müssen, jedoch stets eine notwendige, die zu testen sehr schnell möglich ist.

– Schließlich noch einer Listenstruktur vom Typ `myQueue`, um Referenzen auf die Elemente zu speichern, die das Objekt ausmachen.

Der folgende Code-Ausschnitt beschreibt das in Abschnitt 4.2.1 erwähnte Vergrößern eines Objektes `o` um einen Punkt `p` (Elemente wie Linien und Flächen, die aus 2 bzw. 4 Punkten bestehen, werden später auf diesen Fall reduziert).

```
int distance = (int)((o->center - *p).norm()) -
               o->radius;
if (distance > 0) o->radius += distance;

// difference of border- and point coordinate
int delta;
// is the point right of the right border, but not
// too far away?
if (delta = (p->x - o->xAxis.to->x),
    delta > 0 && delta < maxObjectDistance) {

   // move the leftmost border;
   o->xAxis.to->x    = p->x;
   // correct the center, according to the x-Axis
   o->center.x      += (int)(delta/2);

   // now we have to correct all the x-coordinates
   // of all other Axis! (otherwise, the axis
   // wouldn't cross in the center anymore)
   o->yAxis.from->x += (int)(delta/2);
   o->yAxis.to->x   += (int)(delta/2);
   o->zAxis.from->x += (int)(delta/2);
   o->zAxis.to->x   += (int)(delta/2);
}
// ..... (repeat for the other side of the xAxis;
// afterwards, for y- and zAxis!)
```

- Als letzter Schritt werden alle erkannten Linien, Flächen und Objekte in einer Datei gespeichert, welche von `2show` zum Anzeigen der gesamten Szene eingelesen wird.

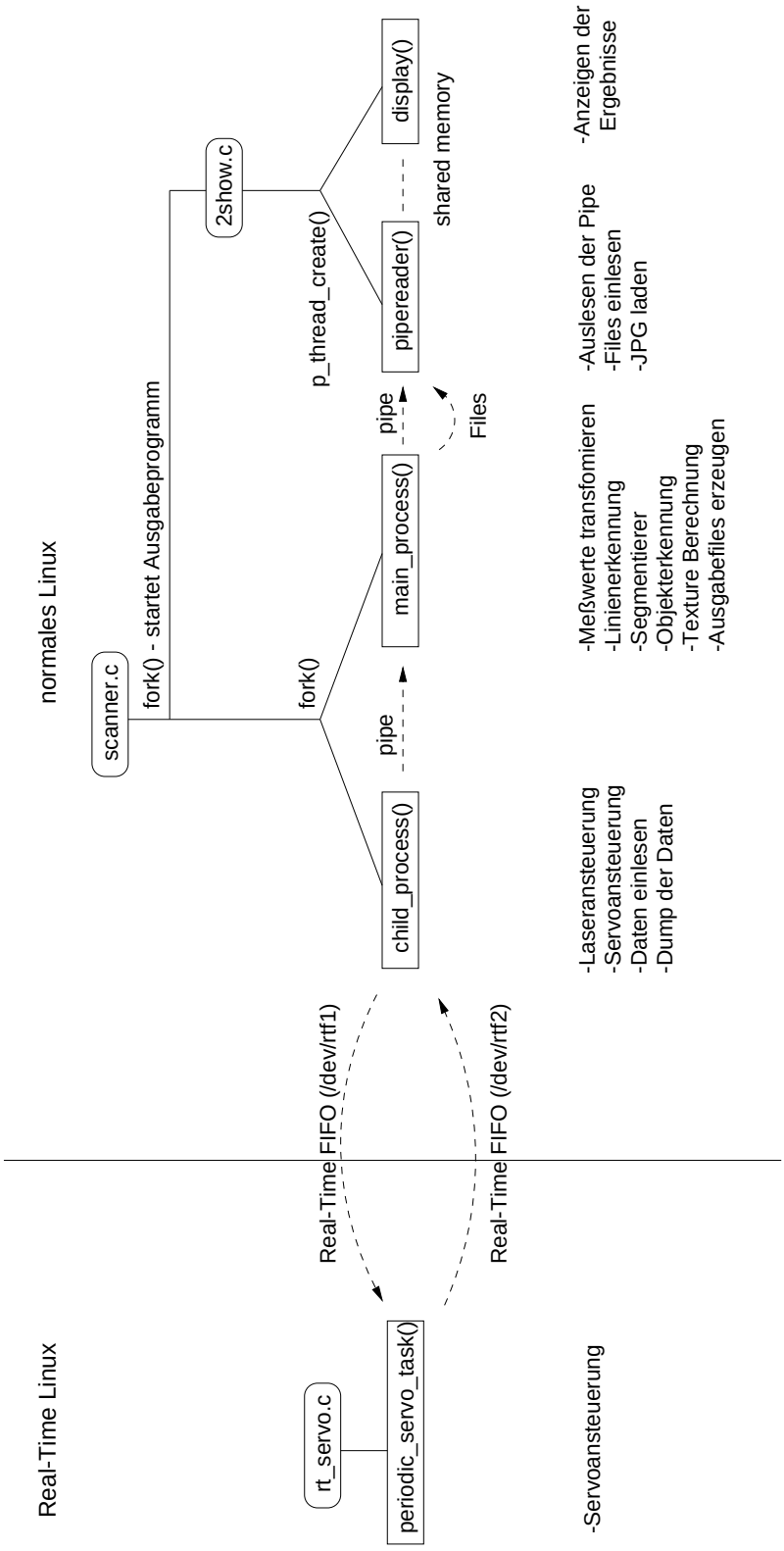


Abbildung 5.1: Überblick über die parallele Architektur der Scanner-Applikation

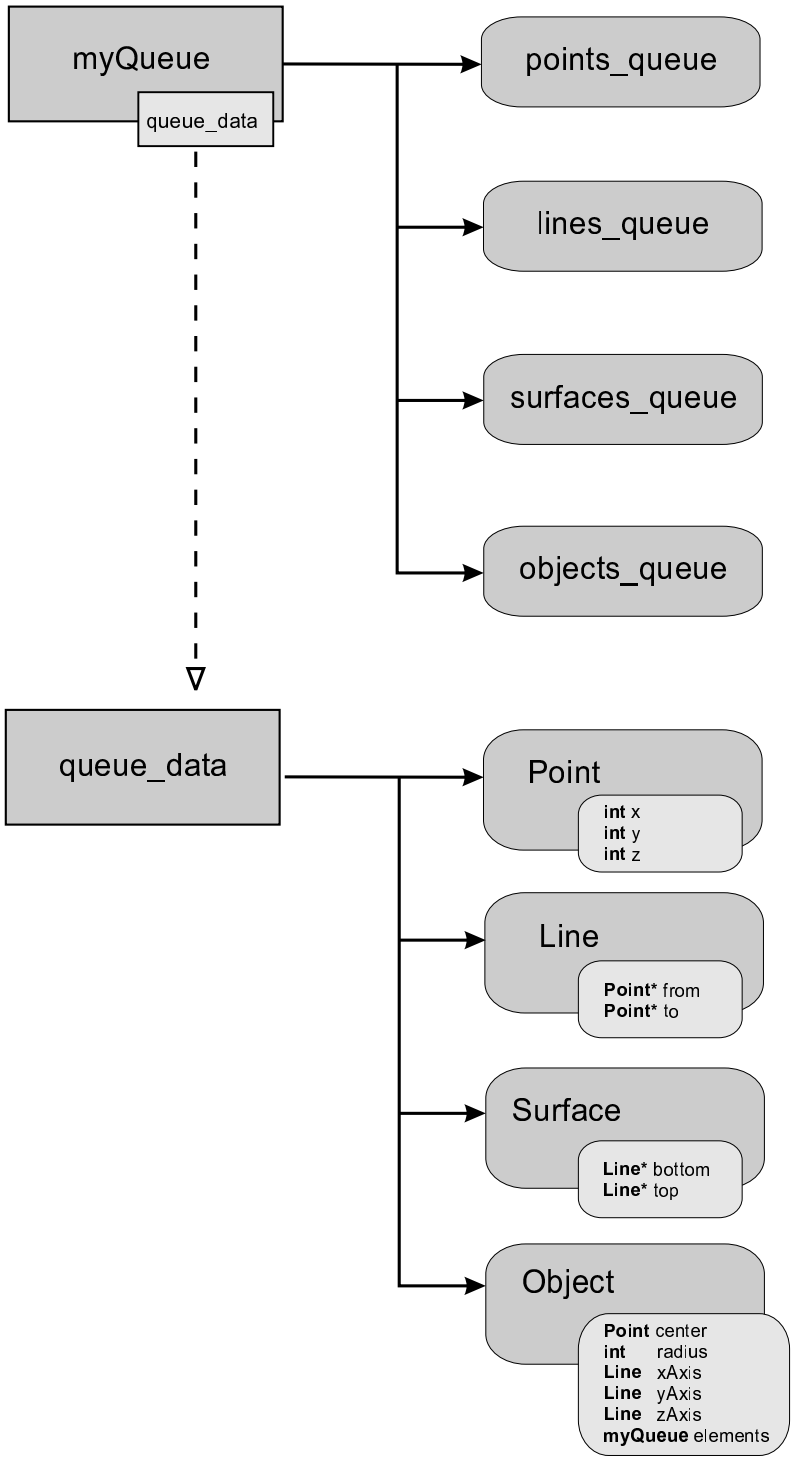


Abbildung 5.2: Klassenhierarchie

5.2 Klassenhierarchie

Einen Überblick über die Klassenhierarchie liefert Abbildung 5.2.

- Wie bereits beschrieben, werden Punkte, Linien, Flächen und Objekte in jeweils einer eigenen Datenstruktur (`points_queue`, `lines_queue`, `surfaces_queue`, `objects_queue`) verwaltet, die jedoch prinzipiell die selben Funktionalitäten aufweisen: diese vier Klassen sind abgeleitet von der Basisklasse `myQueue`, welche großtenteils übliche Listenoperationen bereitstellt.
- Die Basisklasse `myQueue` verwaltet Daten vom Typ `queue_data`, wiederum eine Basisklasse für die vier Element-Klassen `Point`, `Line`, `Surface` und `Object`. Die abgeleiteten Klassen implementieren nun die Datentypen und stellen Element-spezifische Funktionalität bereit. Den besten Überblick darüber liefert die Datei `elements.h`.
  Diese Vererbung ist insbesondere im Fall der Objekt-Liste (der Listenstruktur als Teil der Klasse `Object`) nützlich, da Objekte selber eine Liste als Klassenelement besitzen, welche als Elemente Punkte, Linien und Flächen zu speichern vermag.
- Die vom Scanner gesendeten Daten werden zu Beginn als Pointer auf Instanzen der Datenstruktur `Point` gespeichert. Nahezu alle weiteren Speicherformen (als `Lines` oder `Surfaces`) *referenzieren* lediglich auf diese Punkte, d.h. es existiert keine redundante Mehrfachspeicherung der Daten. Dies geschieht nicht nur aus Speicherplatzgründen, sondern gewährleistet auch, daß der Datenbestand permanent konsistent gehalten wird: es ist nicht notwendig, beispielsweise beim Verschieben eines Punktes alle mit diesem Punkt inzidenten Linien zu aktualisieren.
- Die doppelt verketteten Listen (*Queues*) wurden wiederum gewählt, um den Speicherverbrauch zu minimieren.
  Aufgrund der internen Struktur der Queues ist es trotzdem möglich, einen kompletten Durchlauf der Liste[1] (aller n Element) in Zeit $\mathcal{O}(n)$ anstatt der durchaus üblichen Laufzeit $\mathcal{O}(n^2)$ durchzuführen.

[1]
```
for (int i = 0; i < queue.length(); i++) {
    element = queue[i];
    ...
}
```

5.3 Benutzer-Interface

An dieser Stelle soll das Benutzer-Interface kurz dargestellt werden. Zwar ist es möglich, sowohl das OpenGL– als auch das Scanner–Programm von der Kommandozeile aus zu starten. Die vom Benutzer modifizierbaren Parameter werden dann – wie bereits in Abschnitt 5.1 beschrieben – aus der Konfigurationsdatei `scanner.cfg` gelesen.

Eine sehr viel komfortablere Möglichkeit der Modifikation besagter Parameter und Kontrolle der externen Programme bietet jedoch das graphische Benutzerinterface, geschrieben in der Sprache JAVA.

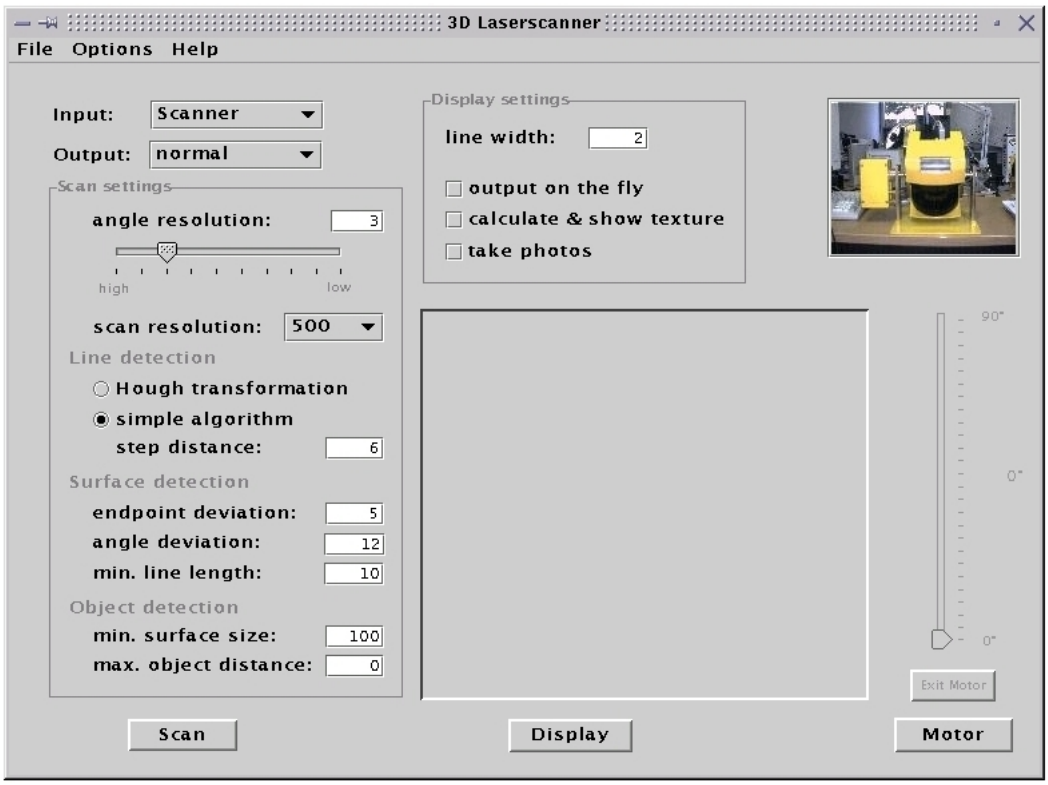


Abbildung 5.3: JAVA-Benutzerinterface: Hauptfenster

5.3.1 Hauptfenster

Das Hauptfenster (vergleiche Abbildung 5.3) ist gegliedert in drei Bereiche, abgeschlossen durch die Buttons *Scan*, *Display* und *Motor*.

Innerhalb des *Scan-Bereiches* lassen sich all jene Parameter einstellen, welche den eigentliche Scan-Vorgang beeinflussen. Die Betätigung des Scan-Buttons startet die Scanner-Applikation unter den spezifizierten Parametern.

Der *Display-Bereich* beinhaltet dementsprechend Parameter, welche die Anzeige beeinflussen. Desweiteren existiert ein Textfenster, in welchem der textuelle Output der Scanner-Applikation ausgegeben wird (siehe auch Abbildung 5.4). Dies sind Informationen, welche die einzelnen Algorithmen als Statusmeldungen dem Benutzer übermitteln, wie beispielsweise die Anzahl erkannter Linien pro Scan. Diese Statusmeldungen werden bei manuellem Start der Scanner-Applikation auf der Textkonsole ausgegeben. Der Display-Button startet das Anzeigeprogramm `2show`, welches den zuletzt durchgeführten Scan visualisiert.

Die Betätigung des *Motor-Buttons* gibt einen Schieberegler frei, mit dessen Hilfe sich die Scanner-Hardware manuell direkt um den eingestellten Winkel drehen läßt.

5.3.2 Parameter

- **Input:**
  Wird die Einstellung *Scanner* gewählt, so werden die Daten aus dem Scanner ausgelesen. Die Einstellung *File* hingegen liest die Daten aus Dateien, welche bei einem früheren 3D-Scan erstellt worden sind.

  Bei der Wahl *File* werden eine Reihe von Parametern (wie beispielsweise die Schrittweite des Motors) deaktiviert, die sich ausschließlich auf das Verhalten der Scanner-Hardware während eines 3D-Scans auswirken.
- **Output:**
  Die Ausgabe läßt sich von einem „*normalen*“ Modus in einen *dump only* Modus schalten, so daß die Daten des Scanners lediglich als Dateien abgespeichert, jedoch nicht weiter verarbeitet werden.

  Hier werden bei der Wahl von *dump only* alle Parameter deaktiviert, die sich auf Algorithmen zur Linien-, Flächen- oder Objekterkennung auswirken. Desweiteren ist es nicht möglich, eine Kombination zu wählen, so daß die Daten aus Dateien gelesen und unbearbeitet als Datei abgespeichert werden.
- **Scan settings:**
  - *angle resolution*:
    Gibt an, in wieviel-Grad-Schritten der Scanner gedreht werden soll, mit 1 Grad als die feinste Einstellung. Der Wert ist wahlweise über den Schieberegler zu wählen oder direkt in dem Textfeld einzugeben.
  - *scan resolution*:
    Der Scanner kann in drei Modi betrieben werden, in denen er wahlweise 500, 250 oder 125 Werte pro Scan liefert.
  - *Hough transformation*:
    Die Hough-Transformation ist die eine von zwei möglichen Linienerkennungs-Algorithmen. Dieser Algorithmus liefert die besseren Ergebnisse als der nachfolgende, ist

jedoch auch langsamer in der Ausführung.

- *simple algorithm*:
  Die Güte dieses zweiten Linienerkennungs-Algorithmus läßt sich über den Wert *step distance* steuern. Es gilt: Je höher der Wert, desto genauer die Abtastung, also die Vektorisierung der diskreten Eingabedaten in Liniensegmente.
- *endpoint deviation*:
  Dieser Parameter entspricht dem Wert ε_1 aus Formel (5.1), der angibt, wie weit die Endpunkte zweier Linien voneinander entfernt liegen dürfen, um von dem Flächenerkennungs-Algorithmus gematcht zu werden.
- *angle deviation*:
  Dieser Wert entspricht ε_2 (Formel (5.3) und gibt den maximalen Winkel zwischen zwei zu matchenden Linien ab.
- *min. line length*:
  Ist die Länge einer Linie geringer als der hier angegebene Wert, so wird sie von dem Flächenerkennungs-Algorithmus ignoriert.
- *min. surface size*:
  Alle Flächen mit mindestens dieser Größe (gemessen in cm^2) werden von vornherein als eigenständige Objekte angesehen. Alle weiteren Elemente werden versucht zu Objekten zu segmentieren.
- *max. object distance*:
  Sind zwei Objekte weiter als die angegebene (euklidische) Distanz voneinander entfernt, so werden sie nicht mehr zu einem einzigen miteinander Objekt verschmolzen.

- **Display settings:**
  - *line width*:
    Gibt die Breite der Linien in Pixel an, die in dem OpenGL–Programm gezeichnet werden.
  - *output on the fly*:
    Wenn aktiviert, wird noch vor dem Scanvorgang `2show` gestartet, so daß das Ergebnis des 3D-Scans schon direkt während der Durchführung nahezu in Echtzeit am Bildschirm beobachtet werden kann.
  - *take photos*:
    Wenn aktiviert, werden während des Scannens vier Photos erstellt, anhand derer das Anzeigemodul Texturen auf die dargestellten Elemente legen kann.
  - *calculate & show textures*:
    Dieser Parameter kann wahlweise deaktiviert werden, um den Scannvorgang zu beschleunigen, da in diesem Fall selbst wenn die Photos geschossen werden die Koordinaten der Texturen nicht berechnet werden.

- **Configuration:** (siehe Abschnitt 5.3.3 sowie Abbildung 5.4)
  - *data directory*:
    In diesem Verzeichnis werden die Daten des Scanners als Dateien abgelegt. Um mehrere 3D-Scans zu speichern ist an dieser Stelle ein anderes Verzeichnis zu wählen (über den „select“-Button).
  - *scanner height*:
    Die Höhe des Scanners ist relevant für Programm-interne Transformationen der einkommenden Daten.
  - *coordinates*:
    Diese vier Koordinaten spezifizieren die maximalen Koordinaten, die von der Kamera abgedeckt werden könne, und sind somit relevant für die Berechnung der Texturen.
  - *device*:
    Das Device, an dem der Scanner angeschlossen ist, ist üblicherweise eine der beiden seriellen Schnittstellen `/dev/ttyS0` oder `/dev/ttyS1`, es kann jedoch auch ein anderes Device angegeben werden.

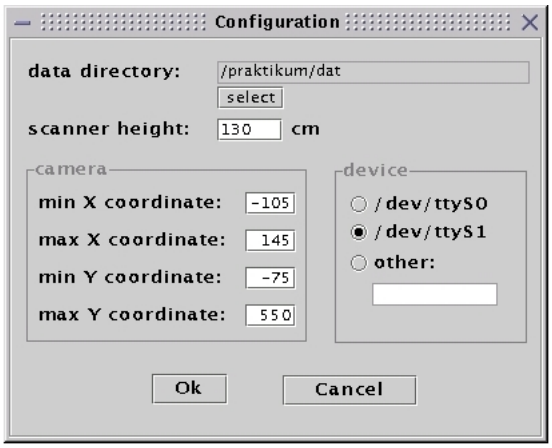


Abbildung 5.4: JAVA-Benutzerinterface: Konfigurationsfenster

5.3.3 Menüstruktur

File Es stehen folgende Untermenüs zur Verfügung:

Load Lädt eine zuvor gespeicherte Konfigurationsdatei.

Save as... Die aktuellen Einstellungen der Oberfläche lassen sich unter einem anderen Namen abspeichern.

Save Sofern zuvor eine Konfigurationsdatei geladen oder der aktuelle Zustand unter einem anderen Namen gespeichert worden ist, so

lassen sich mit diesem Menüpunkt Modifikationen speichern.

Exit Beendet die Oberfläche.

Options Es stehen folgende Untermenüs zur Verfügung:

Colors Die Farben der Oberfläche können neben den Standard-Einstellungen noch in einen Präsentationsmodus gesetzt werden, der sich insbesondere für die Präsentation mittels eines Beamers eignet, sowie in einen Modus, der die Farbe der Scanner-Hardware nachahmt.

Configuration Hier öffnet sich ein weiteres Fenster (vergleiche Abbildung 5.4), in dem die bereits beschriebenen Konfigurations-Parameter einstellbar sind.

Demonstration Der Benutzer kann hier zwischen 4 verschiedenen Demos auswählen (siehe Abbildung 5.5): Bei Betätigung eines der vier Demo–Buttons wird die entsprechende Konfigurationsdatei samt den zugehörigen Datendateien geladen. Der *Input* ist fest auf „File“, der *Output* auf „normal“ gesetzt. Desweiteren wird der Scanvorgang automatisch gestartet.

Sobald das Demonstrationsfenster geschlossen ist, werden die vor dem Demobetrieb geltenden Einstellungen wieder hergestellt, die Applikation kann wie gewohnt bedient werden.

Help Zu guter letzt führt dieser Menüpunkt zu einer kurzen Hilfe-Einblendung.

5.3.4 Implementationsdetails

Im folgenden sei kurz auf die Implementation des Aufrufes der Scanner-Applikation von dem JAVA-Benutzerinterface aus eingegangen.

Das externe Programm `scanner` wird mittels `Runtime.getRuntime().exec()` gestartet. Dabei ist es möglich, die Ausgaben des aufgerufenen Programmes auf den Standardausgabestream abzufangen und als `InputStream` weiterzuverarbeiten: wie in Abbildung 5.5 zu sehen, wird die Ausgabe in das Textfenster der Applikation geschrieben. Aus Performancegründen wird dabei über den `Inputstream` noch ein `BufferedReader` gelegt.

Damit die Ausgabe schon während der Laufzeit des aufgerufenen Programmes angezeigt wird – und nicht erst nach Verlassen der Funktion, wenn die Bildschirmanzeige neu gezeichnet wird – war es notwendig, diese Funktion als eigenständigen *Thread* zu realisieren.

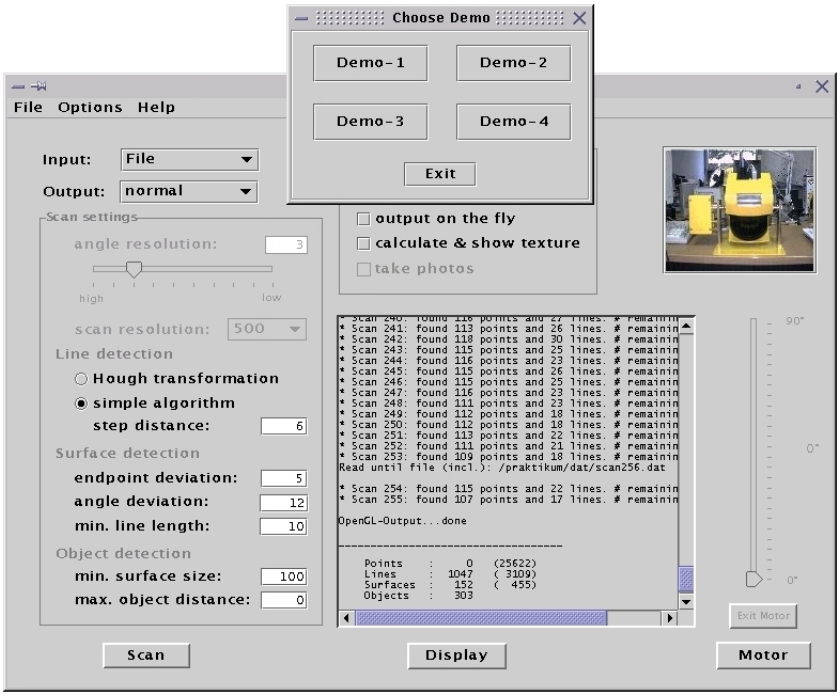


Abbildung 5.5: JAVA-Benutzerinterface: Demonstrationssfenster

```
class startScanner extends Thread {
   Frame father;
   startScanner(Frame f) { father = f; }

   public void run() {
      Process scanner;
      try {
         // save the current values
         father.writeValues(new File(System.getProperty(üser.dir") +
                                    "/bin/scanner.cfg"),true);
         // compose command string
         String exec = "bin/scanner " +
            ("File".equals(father.InputComboBox.getSelectedItem()) ?
            "f" : ("dump only".equals(father.OutputComboBox.
          getSelectedItem()) ? "d" : ß"));

         // start the scanner
         scanner = Runtime.getRuntime().exec(exec);
```

```
            // catch scanner output via 'getInputStream()'
            BufferedReader inStrm = new BufferedReader(
                                    new InputStreamReader(
                                        scanner.getInputStream()));
            String c;
            while ( (c = inStrm.readLine()) != null) {
               father.jTextArea1.append(c + "\n");
            };
         } catch (Exception e) {
            System.err.println("Error: " + e);
            e.printStackTrace();
         }
      }
   }
```

Kapitel 6

3D-Visualisierung

Für die Visualisierung mittels Povray sowie der Mitarbeit an dem OpenGL-Programm danken wir Nicolas Andree.

Im wissenschaftlichen Umfeld bieten sich drei zum Teil grundsätzlich verschiedene Möglichkeiten zur plattformunabhängigen dreidimensionalen Datenvisualisierung an. Diese sind VRML (neu Web3D), OpenGL und Java 3D.

Bei VRML, der Virtual Reality Modelling Language, handelt es sich um ein universelles Dateiformat, welches speziell zum Anzeigen dreidimensionaler Szenen über das Internet entwickelt wurde. Die Anzeige erfolgt durch das Interpretieren von VRML-Dateien durch ein externes Anzeigeprogramm wie z.B. den Cosmo Player. Die Spezifikation von VRML ist auf dessen Homepage einsehbar unter [URLVrm].

Professionelle Software, bei der eine hohe Performance erreicht werden soll, basiert häufig auf OpenGL, dem eine umfassende API zur schnellen 3D Visualisierung zugrunde liegt. Diese Schnittstelle verfügt über ein Repertoire von über 150 Funktionen mit vielfältigen Möglichkeiten der 2D- und 3D-Visualisierung und wird häufig im Rahmen von CAD Anwendungen oder zur Darstellung dreidimensionaler virtueller Welten eingesetzt. Die Tatsache, daß Grafikkartenhersteller inzwischen Mittelklassekarten mit hardwarebeschleunigten OpenGL-Treibern anbieten, erhöht die Darstellungsgeschwindigkeit nachhaltig und somit die Eignung von OpenGL. Die Programmierung erfolgt in C. Die Homepage von OpenGL findet man unter [URLOgl]. Unter Linux steht OpenGL als OpenGL-workalike (jedoch ohne Hardwarebeschleunigung) unter dem Namen „Mesa“ zur Verfügung. Für weitere Informationen siehe auch [URLMes].

Java 3D erlaubt es dem Entwickler, seine Applikation im Rahmen der Java Virtual Machine komplett objektorientiert zu entwickeln. Ausgangspunkt der Modellierung von dreidimensionalen Szenen ist hier immer der Szenengraph. Als low-level Schnittstelle zur eigentlichen dreidimensionalen Darstellung wird entweder OpenGL oder aber Direct3D von Microsoft benutzt. Die Programmierung mit Java 3D erfolgt objektorientiert und ist plattformunabhängig. Spezifikationen aller Klassen, weitere Informationen und Tutorien sind auf der Java 3D Homepage unter [URLJ3D] abrufbar.

6.1 Erste Visualisierung der Testdaten

Erste Bilder, die einen Einblick in die Güte der zu erwartenden Ergebnisse liefern sollten, entstanden aufgrund der Vertrautheit mit dem Umgang mit dem Raytracer Povray. Diese Bilder sahen bereits recht vielversprechend aus, vergleiche Abbildung 6.1.

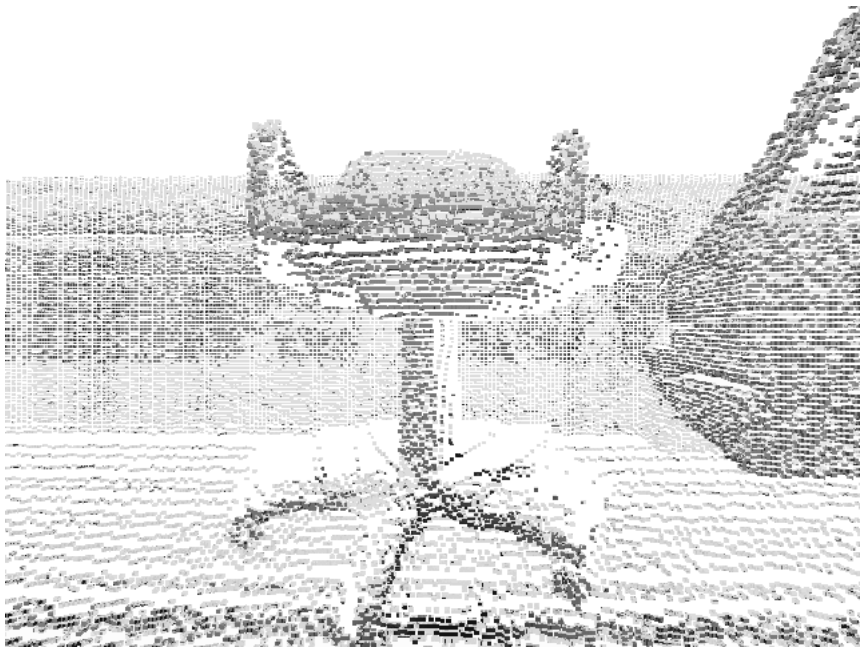


Abbildung 6.1: Erste Darstellung mittels Povray

Die Visualisierung mittels Povray unterliegt jedoch folgenden Nachteilen:

1. Die generierte Beschreibung der Szene muß kompiliert werden. Dieser Vorgang benötigt bei etliche tausend Meßpunkten, die jeweils einzeln durch eine Povray-Primitive repräsentiert werden, mehrere Stunden.

2. Das generierte Sourcefile erreicht leicht eine Größe von mehreren Megabyte.

3. Die gerenderte Szene ist statisch. Um die Szene aus einem geänderten Blickwinkel betrachten zu können, ist es notwendig, die Datei neu zu kompilieren.

Daher haben wir Povray ladiglich als erste, einfache Möglichkeit verwendet, um Scannerdaten zu visualisieren und damit die Machbarkeit des hier vorgestellten 3D-Scanners zu beurteilen.

Für die eigentliche Darstellung der Scannergebnisse benutzen wir OpenGL.

6.2 Die Datenstrukturen innerhalb des OpenGL-Programmes

Aus Gründen der Performance sowie aufgrund der einfachen Handhabung in C werden für Datenstrukturen zur Speicherung von Objektdaten wie Koordinaten, Texturdaten, etc. Felder verwendet. Diese werden im Shared Memory abgelegt.

Die Speicherallokation dieser Felder lies sich mit Hilfe der `realloc()`-Funktion auf eine dynamische Art und Weise realisieren. Dies ist besonders deswegen hervorzuheben, weil bei einer Online-Darstellung des Scannvorganges die Anzahl der eingescannten Meßwerte und damit den benötigten Speicher nicht bekannt ist. Der Speicherbedarf ist enorm, wenn der Scanner in der höchsten Auflösung betrieben wird (113400 Punkte – vgl. Tabelle 3.2). Zu jedem Punkt werden 3 Integer (Koordinaten) und bis zu 4 Floats (Texturkoordinaten, Farbe) abgespeichert.

6.3 Benutzerinteraktion

Es wurde eine freie Navigation innerhalb der visualisierten Szene implementiert. Zur Verfügung stehen Bewegungen des Blickpunktes der virtuellen Kamera in alle drei Richtungen des Koordinatensystems, was durch Transformationen realisiert wird. Desweiteren gibt es Rotationen um die *x*–, *y*– und *z*–Achse. Quelltexte findet man in Anhang C.3.1.

6.3.1 Main Eventloop

Wie in fast jedem OpenGL Programm findet man auch hier die beiden Funktionen `glutKeyboardFunc(Key)` und `glutDisplayFunc(display)` in der `main()`-Funktion. Diese Funktionen befinden sich in einer Eventloop, der `glutMainLoop()`. Diese ermöglicht es, daß die Änderungen von Statusvariablen unmittelbar in die entsprechende Transformation oder Rotation umgesetzt werden. Mithilfe der Funktion `glutReshapeFunc(Reshape)`, die sich ebenfalls im `glutMainLoop()` befindet, wird dann die Szene neu gezeichnet.

6.4 Übersicht der Funktionalität

In der Scannerapplikation stehen nun die folgenden Visualisierungs-Modi zur Verfügung:

- Anzeigen aller Meßwerte als Punkte,
- Verbinden der Punkte zu einem Drahtgittermodell, sowie die
- Anzeige der Ergebnisse der digitalen Bildverarbeitung. Dazu gehören insbesondere die in der Szene erkannten Linien, Flächen und Objekte.

In jedem dieser Modi kann man sich durch die Szene bewegen, d.h. man kann die Szene aus verschiedenen Perspektiven betrachten.

Das Umschalten zwischen den einzelnen Modi wird durch Betätigen der entsprechenden Tasten realisiert. Hier zunächst eine Übersicht der Darstellungsformen und der zugehörigen Tastaturbelegung:

Punkte	Linien	Flächen (Surfaces)	Objekte
'p'	'l'	's'	'o'

Drahtgitter	Gittergröße	Gittersegmentierung	Gittersegmentierungsgröße
'q'	'+', '−'	'w'	'÷', '×'

Für die Anzeige stehen grundsätzlich zwei Modi zur Verfügung: Die Objekte können entweder mit Texturen versehen oder gemäß ihres Typs eingefärbt werden. Polygone werden wahlweise offen oder geschlossen dargestellt. Es ist jeweils möglich, zwischen diesen hin- und herzuschalten.

Polygone offen/gefüllt	Texturen	Grundebene zeichnen	Reset	Springen	Hilfe
'Space'	't'	'b'	'r'	'n'	'h'

Folgende Funktionen sind zur Navigation in der Szene gedacht. Mit Benutzung des numerischen Tastenblocks ist eine recht intuitive Bewegung möglich. Die Rotationen beziehen sich immer auf den Koordinatenursprung.

Rotation um die x-Achse	Rotation um die y-Achse	Rotation um die z-Achse
$\uparrow, \downarrow$	$\leftarrow, \rightarrow$	$7, 1$

Translation in x-Richtung	Translation in y-Richtung	Translation in z-Richtung
$4, 6$	$8, 2$	$9, 3$

6.4.1 Punktedarstellung

Es gibt zwei alternative Arten von Punktvisualisierungen.

- Zum Anzeigen *aller* Meßpunkte wird die OpenGL-Primitive `GLPoints` benutzt. Um dies zu ermöglichen, werden die Punkte aus der Pipeline oder aus der Datei `Points.ogl` importiert. Intern sorgt der Aufruf der Funktion `DrawPoints()` für das Ausführen der Darstellung.
- Zur Visualisierung der *reduzierten* Datenmenge wie in Kapitel 3.2.1 beschrieben wird auf die OpenGL-Primitive `GLQuads` zurückgegriffen, Punkte werden somit als kleine Flächen dargestellt. Dies geschieht mit Hilfe der Funktion `DrawQuads()` (vgl. Anhang C.3.2 für den Quelltext).

  Die Funktion `DrawQuads()` ist allgemein gehalten, kann daher bei anderer Parametrisierung auch für die Darstellung von Polygonen wie Flächen, Objekten und Linien (auch

dargestellt als Polygone) verwendet werden. Es stehen zwei Polygondarstellungsmodi „*Polygone offen*“ und „*Polygone gefüllt*“ zur Auswahl. Die Darstellung durch offene Polygone ist sinnvoll, um ansonsten verborgene Punkte visualisieren zu lassen.

6.4.2 Linien-, Flächen- und Objektdarstellung

Zum Anzeigen von Linien und Flächen gibt es wieder je zwei Modi, die die Abhängigkeit zeigen, wie weit beispielsweise Linien in der nächsten Stufe der Bildverarbeitung durch Flächen oder Objekte gematcht werden. Insgesamt kommen die in der Tabelle verzeichneten Fälle vor:

Linien	Flächen	Objekte
alle erkannten	alle erkannten	alle erkannten
nicht zu Flächen oder Objekten gematchte	nicht zu Objekten gematchte	%

Die Idee dabei ist, daß die in der nächsten Stufe der Bildverarbeitung nicht gematchten Linien beispielsweise als Information beim Aktivieren des Anzeigens der nächsten Stufe (z.B. die Flächen) zusätzlich sichtbar gemacht werden können. Diese Information steht dem Betrachter also auch weiterhin zur Verfügung. Die einzelnen Linien, Flächen und Objekte werden nun wieder von der Funktion `DrawQuads()` gezeichnet.

6.4.3 Drahtgittermodell

Die Wahl des Modes der Polygondarstellung spielt auch bei der Drahtgitterdarstellung eine Rolle, da sie mit Hilfe der OpenGL-Primitive `GLQuadStrip` realisiert wurde.

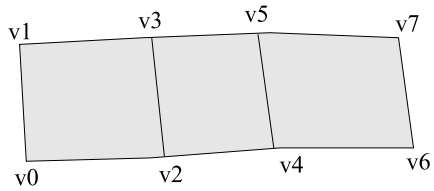


Abbildung 6.2: Funktionsweise der OpenGL-Primitive `GLQuadStrip`.

Primär ist die Funktion `DrawQuadStrip()` für die Drahtgitterdarstellung verantwortlich. Mittels den Tasten ’+’, ’-’ knan die Grösse des Drahtgitters angepasst werden.

Im Modus „*Polygone geschlossen*“ entsteht eine homogene Fläche. Aus diesem Grunde wird das Textur-Mapping von dem Mode „*Polygone geschlossen*“ natürlich begünstigt, da keine Lücken mehr auftreten können, wie dies z.B. bei einer reinen Punktedarstellung noch möglich ist.

6.4.4 Segmentierung im Drahtgittermodell

Man kann sich überlegen, daß Objekte im Drahtgittermodell durch einen langgestrekten QuadStrip verbunden sind. Dies liegt daran, daß die Tiefeninformation stark schwankt.

Die Idee ist nun, diese Eigenschaft auszunutzen, um einen einfachen Segementierer zu programmieren. Dabei werden einfach entartete Quadstrips, also Quadstrips, mit zu langen Diagonalen, nicht gezeichnet. Mit 'w' läßt sich ein solcher einfacher Segmentierer ein/ausschalten, mit '$\div$' und '$\times$' kann man den Schwellenwert für die Länge der Diagonalen variieren. Abbildung 6.3 zeigt einen Ausschnitt eines Scans mit und ohne Segmentierung (der Blickpunkt wurde etwas zur Seite verschoben, damit man die Tiefeninformation besser sieht). Man erkennt, daß die Umrisse der gescannten Person klar zu sehen sind. Weitere Bilder befinden sich in Abschnitt 6.6.

Mit diesem Ansatz alleine lassen sich jedoch naturgemäß nicht alle Objekte korrekt segmentieren. So sind zum Beispiel lange Kanten eines Objektes, die parallel zur Laserrichtung verlaufen, problematisch. Weiterhin ist bisher nicht berücksichtigt, daß weit entfernte Scanpunkte einen größeren Abstand zu den Nachbarpunkten aufweisen. Diese werden demzufolge auch segmentiert.

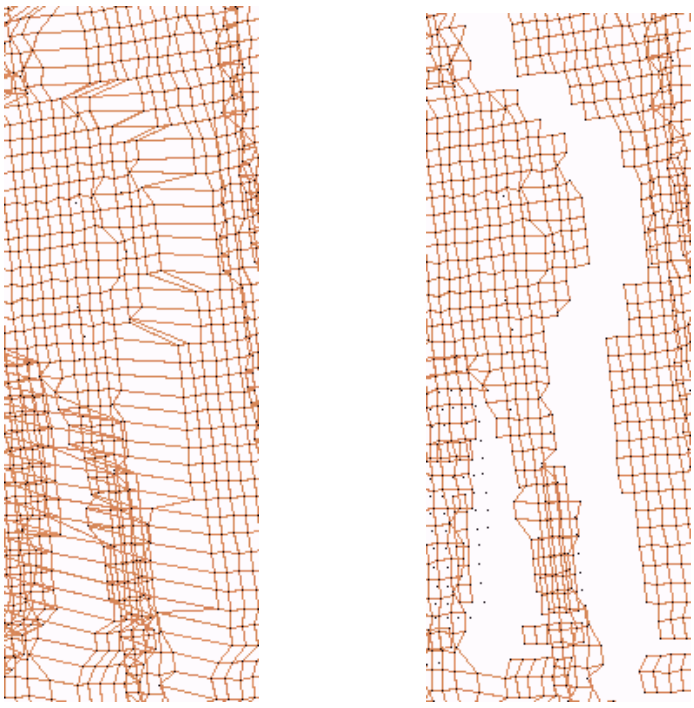


Abbildung 6.3: Die Drahtgittersegmentierung

6.5 Datenimport

Es gibt die Möglichkeit, die Daten bereits während des Scanvorgangs im Online-Anzeigemodus aus der Pipeline einzulesen und anzuzeigen. Der Import der eingescannten Daten erfolgt dann entweder direkt aus einer Pipeline oder aus den beiden Dateien `Data.ogl` und `Points.ogl`.

6.5.1 Online-Anzeige

Um die Online Anzeige zu ermöglichen, ist es notwendig, die Einlesefunktion als einen weiteren Prozeß innerhalb des Anzeigeprogramms zu deklarieren. Aus der Scanner-Pipeline werden die Daten in die entsprechenden Felder kopiert und parallel durch die Display Funktionen angezeigt. Die Felder befinden sich im Shared Memory. Während dieses Einlesens wachsen die Felder inkrementell. Bei der Online-Ausgabe wird, um bessere Performance zu erhalten, die OpenGL Darstellung nur aller 2 Sekunden aktualisiert.

Falls die Daten bereits vorliegen und kein Online-Betrieb durchgeführt wird, können die Daten aus den Dateien `Data.ogl` und `Points.ogl` eingelesen werden. Für die Daten aus `Points.ogl` ist es ebenfalls möglich, den schrittweisen, dynamischen Aufbau der Szene Scan für Scan wie im Online-Modus mitzuverfolgen.

Gleichzeitig erfolgt der Import der Daten, die in vorigen Programmteilen bearbeitet wurden, aus der Datei `Data.ogl`. Diese Aufgabe erfüllt die Funktion `read_file()`. Dieser Import umfaßt alle Linienkoordinaten, Punkte, die nicht auf erkannten Linien liegen, erkannte zusammenhängende Flächen sowie die Ergebnisse der Objekterkennung.

6.5.2 Texturen in OpenGL

Das OpenGL Anzeigeprogramm liest die jpeg-Bilder der Größe 506×997 in ein RGBA-Array der Größe 1024×2048. Dabei werden mehrere Fotos zu einem Bild zusammengefügt. Das etwas größere Array wurde zuvor mit einem einheitlichen Farbwert (Hellgrau) gefüllt. Das Einlesen geschieht derart, daß die linke untere Ecke des Bildes den Arrayindex (1,1) bekommt. Dadurch entsteht ein 1 Pixel breiter Rand an der linken und unteren Bildkante.

OpenGL stellt einfach zu handhabende Routinen zur Textur-Verarbeitung zur Verfügung. Jedem Punkt (`glVertex3f`) kann eine Textur-Koordinate mit dem Befehl `glTexCoord2f` zugeordnet werden. Das Mapping der Texturen geschieht dann automatisch. Die Argumente der Funktion `glTexCoord2f` sind zwei Floatingpoint-Zahlen zwischen 0 und 1, die die Position in dem Textur-Array spezifizieren. Falls bei dem Initialisierungsbefehl `glTexParameteri` die Einstellung `GL_CLAMP` angegeben wurde, wird bei Textur-Koordinaten außerhalb des Bereiches (0,1) das jeweils letzte Pixel innerhalb der Textur kopiert. Dies erklärt den ein-Pixel breiten Rand um die eigentliche Textur. Dadurch wird sichergestellt, daß alle Bereiche, von denen keine Textur-Informationen vorliegen, einheitlich eingefärbt werden.

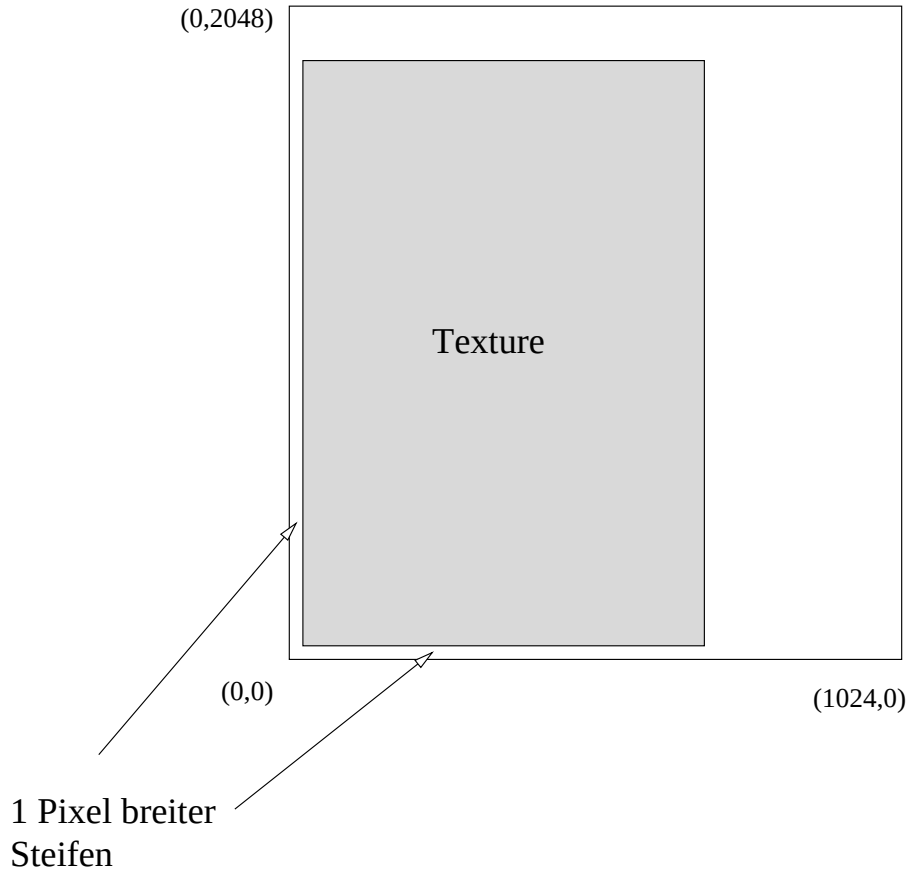


Abbildung 6.4: Die Texture in OpenGL

6.5.3 Aufbau der Datenaustausch-Dateien

Zunächst die BNF für grundlegende Daten und Objekte:

```
<Int>          ::= <IntDigit> | +<IntDigits> | -<IntDigit>.
<IntDigits>    ::= ‘1’ | ‘2’ | ... | ‘0’ | <IntDigits>
                   <IntDigits>.
<Double>       ::= <DoubleDigits> | +<DoubleDigits> |
                   -<DoubleDigits>.
<DoubleDigits> ::= <IntDigits> | <IntDigits>.<IntDigits>. |
                   .<IntDigits>.

<Objekt>       ::= <Fläche> <NL> <Fläche>.
<Fläche>       ::= <Linie> <NL> <Linie>.
<Linie>        ::= <Punkt> <NL> <Punkt>.
```

```
<Punkt>        ::= <Int> <K> <Int> <K> <Int> <K> <Double>
                   <K> <Double>.

<K>            ::= ‘,’.
<NL>           ::= ‘\n’.
```

Syntax von `Points.ogl`

Die Datei `Points.ogl` beinhaltet alle eingescannten Meßpunkte. Der Aufbau der Datei ist sehr einfach. Jede Zeile enthält genau einen Meßpunkt. Die Speicherung folgt der Reihenfolge, in der die Daten eigescannt wurden, so daß die vorgegeben Ordnung erhalten bleibt.

Der Integerwert in der ersten Zeile gibt an, wieviele Punkte sich jeweils in einem Scan befinden. Die Datei hat somit folgende Struktur:

```
<File>         ::= ‘0’ | <Int> <NL> <Points>.
<Points>       ::= <Punkt> | <Punkt> <NL> <Points>.
```

Syntax von `Data.ogl`

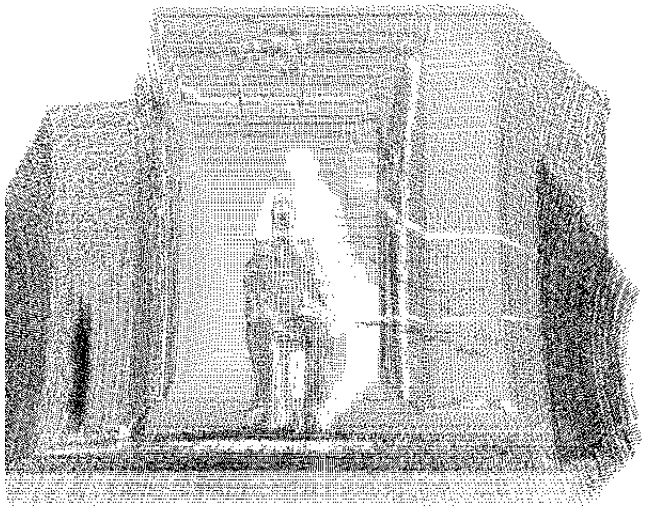
Zu Beginn steht die Anzahl der jeweils von diesem Typ anzuzeigenden Werte sowie ein String, um welchen Typ es sich handelt. Als Typ stehen „Points“, „Lines“, „Surfaces“ und „Objects“ zur Auswahl. Das bedeutet, wenn beispielsweise der String „Surfaces“ gelesen wird, alle nachfolgenden Daten (bis zu dem nächsten typisierenden String) als Flächen nach der angegebenen Syntax zu interpretieren sind.

```
<File>         ::= <Points> <NL> <Lines> <NL> <Surfaces>
                   <NL> <Objects>.
<Points>       ::= ‘0 Points’ | <Int> ‘Points’ <NL> <P>.
<P>            ::= <Punkt> <NL> <P> | <Punkt>.
<Lines>        ::= ‘0 Lines’ | <Int> ‘Lines’ <NL> <L>.
<L>            ::= <Linie> <NL> <L> | <Linie>.
<Surfaces>     ::= ‘0 Surfaces’ | <Int> ‘Surfaces’ <NL> <S>.
<S>            ::= <Fläche> <NL> <S> | <Fläche>.
<Objects>      ::= ‘0 Objects’ | <Int> ‘Objects’ <NL> <O>.
<O>            ::= <Objekt <NL> <O> | <Objekt>.
```

Die vollständige Beschreibung eines Punktes besteht hier also aus drei Integern für die Raumkoordinaten im $\mathbb{R}^3$, zwei Floats für die Textur-Koordinaten und drei weiteren Floats für eine alternative Farbgebung. Zur Beschreibung einer Linie werden zwei Punkte, zur Beschreibung von Flächen vier Punkte und zur Darstellung von Objekten acht Punkte benötigt. Für einen beispielhaften Aufbau der Dateiformate siehe auch Anhang C.4.

6.6 Screenshots des 3D-Outputs

Auf den nachfolgenden Seiten sind nun einige Screenshots von Beispiel-Scans zu sehen: die Bilder sind bei einem Gang durch ein Gebäude aufgenommen worden und zeigen anschaulich die aktuellen Möglichkeiten des vorgestellten 3D-Laserscanners.



Anhand dieser ersten Szene werden im folgenden die verschiedenen Visualisierungsmöglichkeiten dargestellt. Zunächst sind die reinen Scanpunkte zu sehen, die von der 3D-Scannerhardware geliefert werden. Bemerkenswert ist die optisch schon recht gute Qualität, die sich schon nach einer simplen Koordinatentransformation (siehe Abschnitt 3.2.2) ergibt.

Die – gerade an den Seiten – sichtbare Krümmung stammt von der Drehbewegung des Scanners um seine X–Achse. Offentsichtlich beschreibt der Auftreffpunkt des Lasersstrahls eine halbkreisförmige Bahn auf einer in Blickrichtung des Scanners verlaufenden Ebene.

Auf der rechten Seite des Bildes sind zwei horizontale Störungen zu sehen, d.h. die Wand erscheint an diesen Stellen unterbrochen und die zugehörigen Scanpunkte sind nach vorne versetzt. Dies entstand durch zwei Personen, die während des Scannens duch den Flur gelaufen sind. Die dabei auftretenden Veränderungen sind in Abbildung 6.5 verdeutlicht.

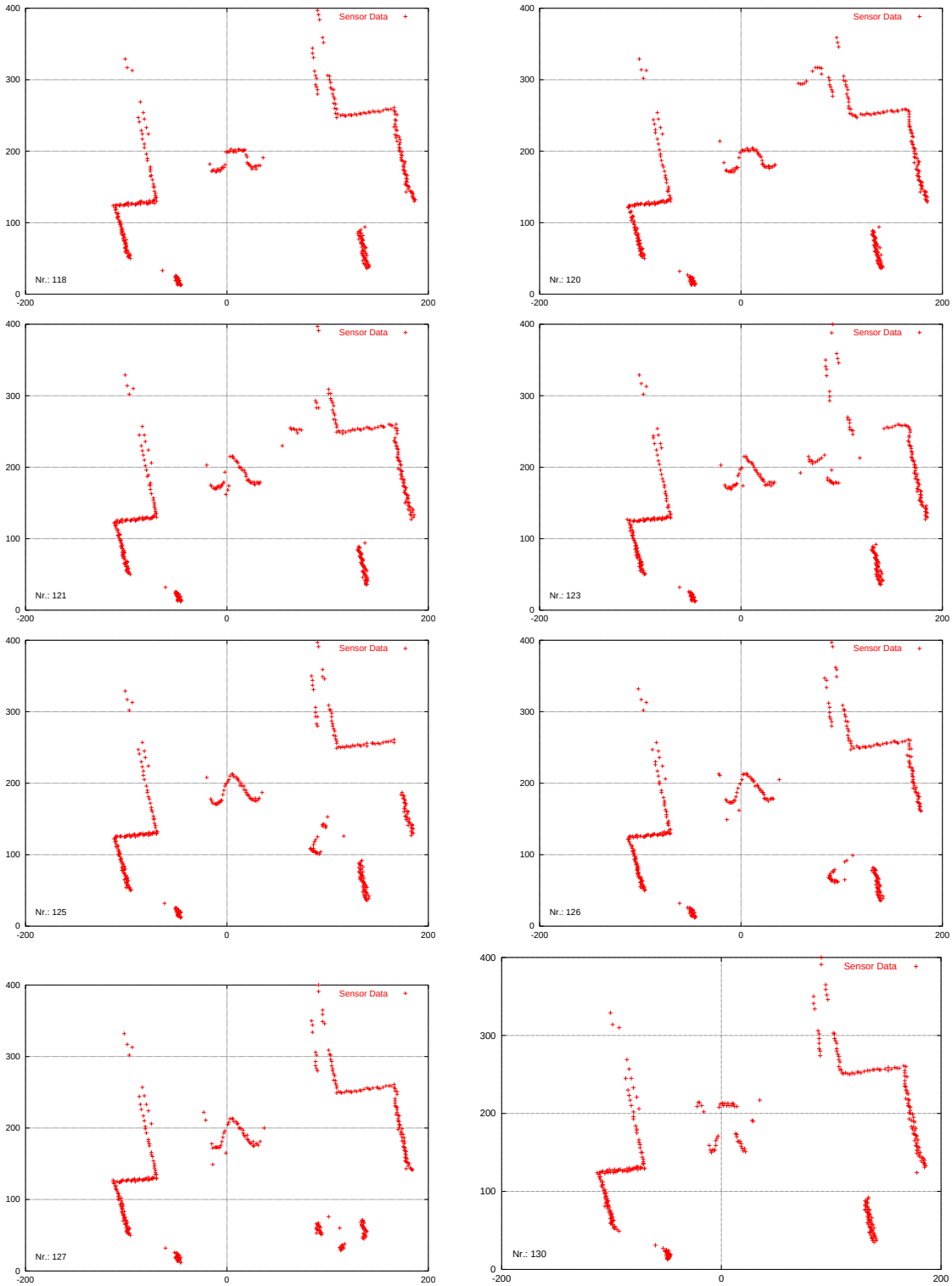
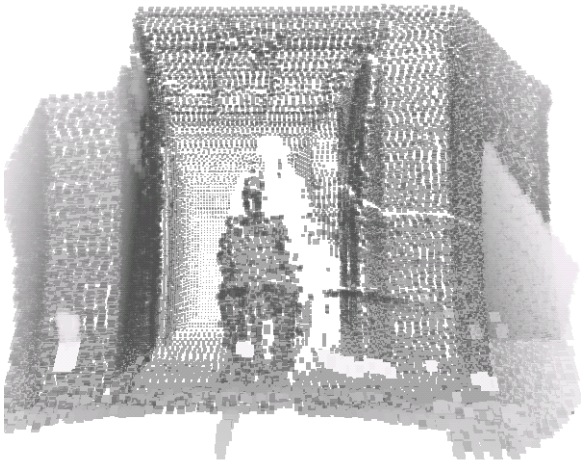
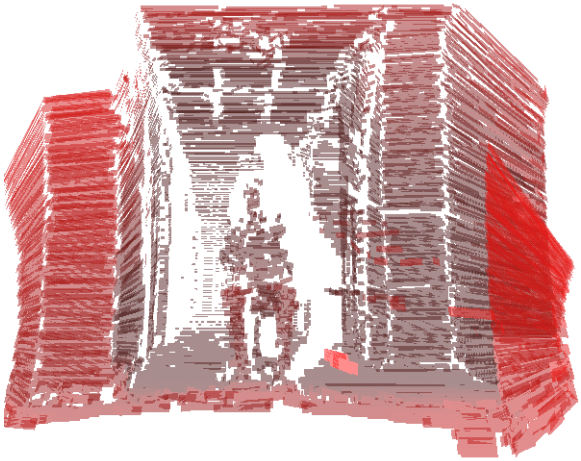


Abbildung 6.5: Vorbeilaufen eines Menschens auf der rechten Seite (von hinten kommend)

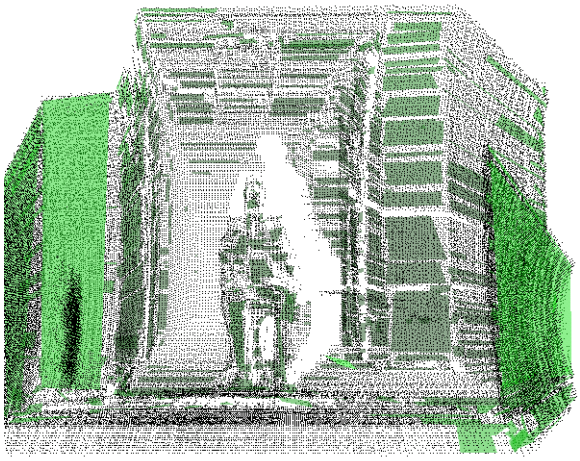


Im Gegensatz zur reinen Punktedarstellung des letzten Bildes erlaubt die in Abschnitt 3.2.1 geschilderte Reduktion der Daten, diese als kleine Vierecke mit konfigurierbarer Größe darzustellen, ohne auf akzeptable Performance verzichten zu müssen.



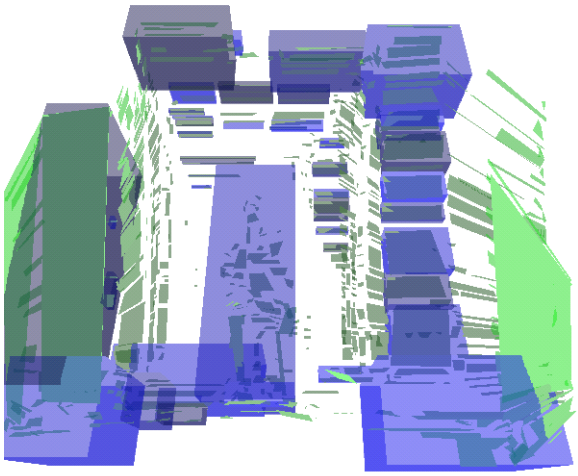
Die folgenden Bilder visualisieren die Ergebnisse des Linienerkennungs-Algorithmus. Dabei mußten empirische Schwellwerte so eingestellt werden, daß auch recht kurze Linien als solche erkannte werden (vergleiche nächste Bilder).

Die erkannten Linien werden zu Flächen zusammengefaßt. Schön zu sehen ist die große Fläche auf der linken Seite. Die Person in der Mitte ist nun als ein lose zusammemnhängender Haufen unterschiedlich orientierter, kleinerer Flächen sichtbar.

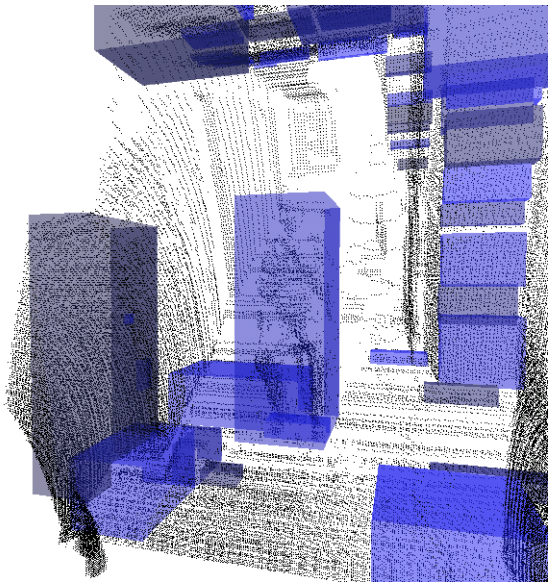


Legt man die erkannten Flächen über die Scandaten, läßt sich anschaulich die Korrektheit des Flächenerkennungs-Algorithmus plausibel machen. Die Reihe unzusammenhängender Flächen auf der rechten Seite haben Ihren Ursprung in den bereits erklärten Störungen durch Personen, die durch die Szene gelaufen sind. Dies soll als Beispiel dafür dienen, daß die hier vorgestellten Algorithmen sehr robust gegen diese Art von Umgebungsveränderungen ist.

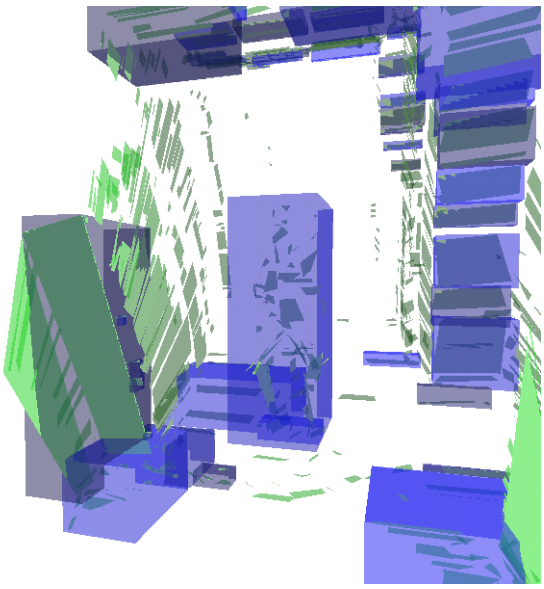
Die aus diesen Flächen entstehenden Objektbegrenzungen reichen trotz der Bilstörungen zur sicheren Hindernis-Vermeidung aus.



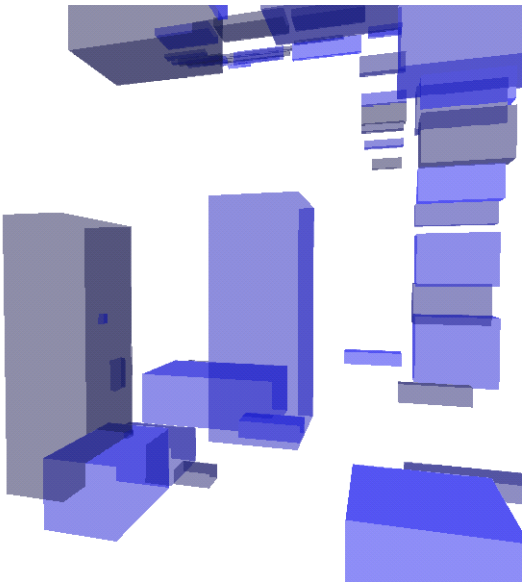
Größere Flächen wie die linke Wand werden als eigenständige Objekte betrachtet, kleinere Flächen werden zu größeren Objekten zusammengefügt, sofern sie hinreichend eng beieinander liegen. Die Objekte werden durch die sie begrenzenden Boundingboxes dargestellt.



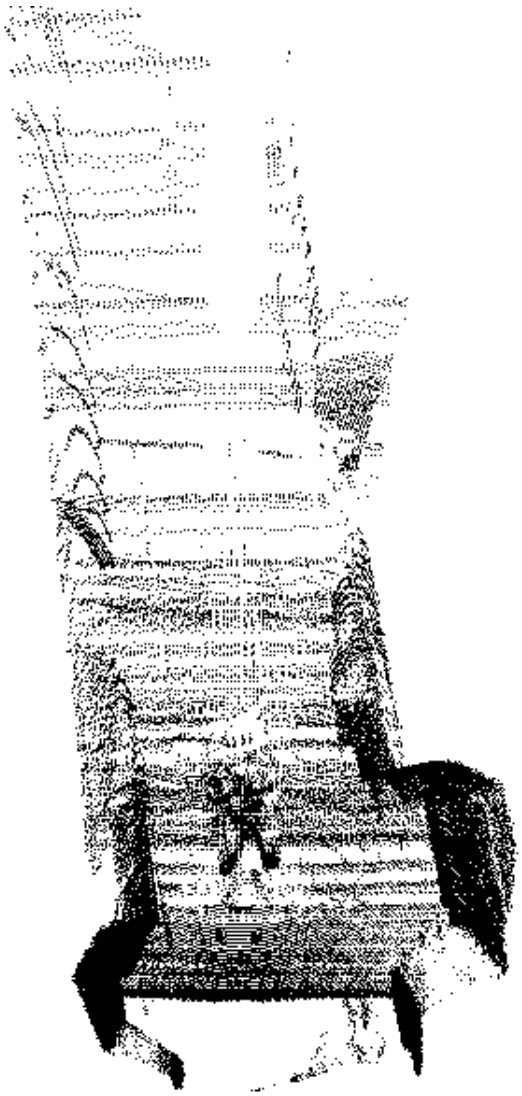
In dieser Ansicht sieht man sehr gut, wie die Person in der Mitte vollständig von einer Box umschlossen wird.



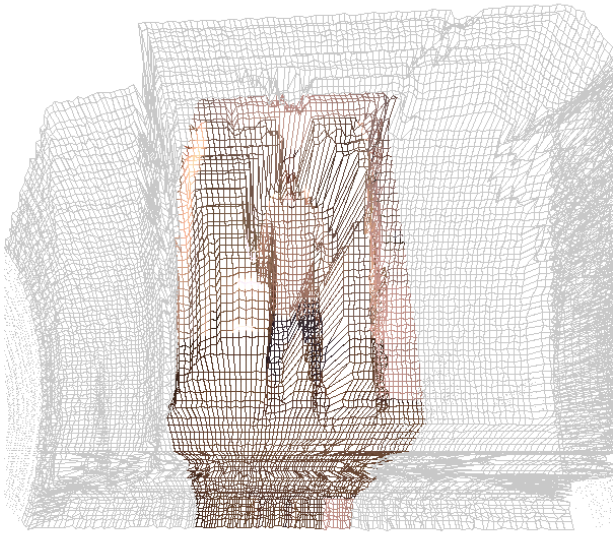
In dieser etwas geänderten Ansicht ist die dreidimensionale Struktur der Szene sichtbar. Man erkennt, daß die Elemente (und insbesondere die Person) wirklich durch dreidimensionale, abgeschlossene Objekte repräsentiert werden.



Solch eine reine Objektdarstellung eignet sich als eine dreidimensionale Belegtheitskarte des gescannten Raumes, beispielsweise zur Roboternavigation.

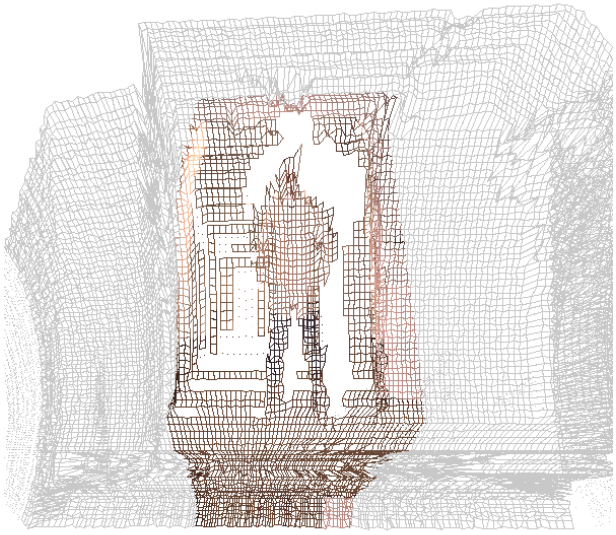


Darstellung der gescannten Szene von oben. Im Gegensatz zu einfachen 2D-Laserscannern – mit denen eine ähnliche Darstellung auch zu erreichen gewesen wäre – ist es hier möglich, Objekte wie Boxen, Eimer, Stühle etc. aufgrund ihrer geringen Höhe und Ausmaße als Einrichtungsgegenstände des Raumes zu identifizieren. Zweidimensionalen Scannern fällt es schwer, zwischen dem Raum selber und solchen Gegenständen zu unterscheiden, so daß selbst in solch einfachen Kartographie-Einsätze der 3D-Laserscanner eindeutig von Vorteil ist.



Eine weitere Möglichkeit der Darstellung ist das Drahtgitter: Dazu werden Scandaten (mit veränderbarer Auflösung quantifiziert) als Punkte genommen, die untereinander verbunden werden (vergleiche auch Seite 87).

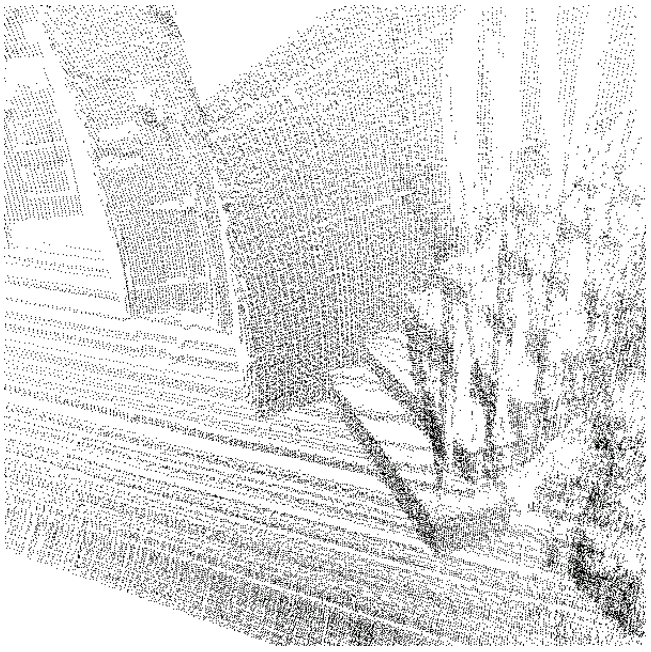
Ferner kann das Drahtgitter mit einer Textur belegt werden, die aus den während des Scans aufgenommenen Fotos berechnet wird. Gescante (Rand-)Gebiete, die nicht von den Fotos abgedeckt werden, sind grau dargestellt.



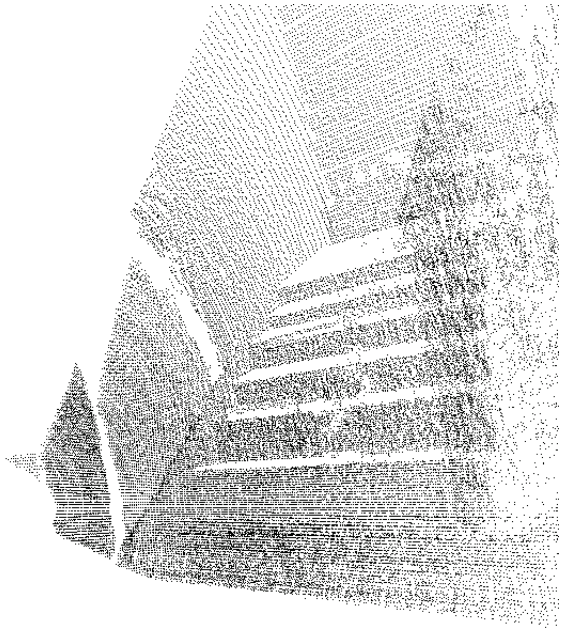
Mithilfe des Drahtgitters bietet sich ebenfalls eine einfache Möglichkeit der Segementierung: Hierzu werden die Facetten des Drahtgitters nur dann gezeichnet, wenn ihre Diagonalen einen Schwellwert nicht überschreiten.

Dieser Schwellwert läßt sich interaktiv anpassen, das Resultat ist eine Segmentierung der Person in der Mitte des Ganges.

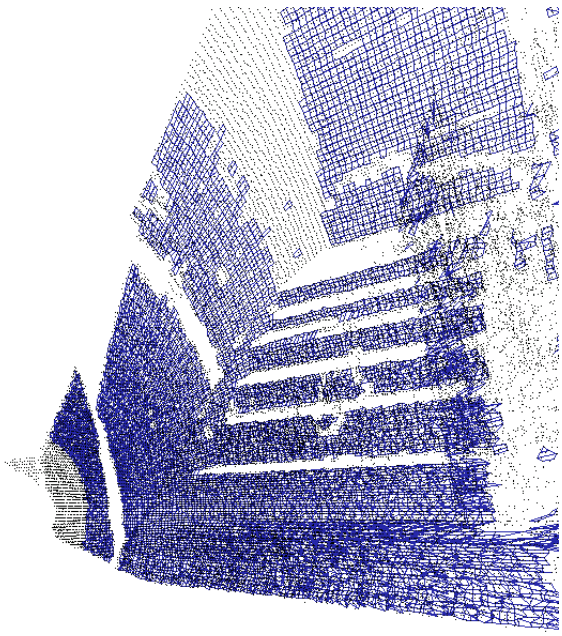
Werden nun die Facetten des Gitters ausgefüllt gezeichnet, lassen sich die segmentierten Objekte komplett mit ihren Texturen überziehen.



Ein neues Szenario: Aufnahmen einer Treppe. Rechts ein Foto der Szene, zusammengesetzt aus drei Kamera-Aufnahmen. Links das Ergebnis eines 3D-Scans.



Anhand dieser frontalen Ansicht der Treppe läßt sich gut die Fähigkeit des Drahtgitter-Segmentierers aufzeigen.



Es ist möglich, selbst die einzelnen Treppenstufen als voneinander getrennte Bereiche zu erkennen.

Kapitel 7

Zusammenfassung und Ausblick

Wichtige Fragestellungen der Robotik wie beispielsweise die Kartierung und Navigation aber auch Andock- und Einparkaufgaben erfordern eine Reaktion eines Roboters in Echtzeit. Dazu benötigen die Roboter präzise, schnelle und preisgünstige Sensoren, die trotz des begrenzten Vorrats an Energie und Rechenleistung den Roboter in die Lage versetzen, seine Umgebung effektiv und effizient zu erfassen. In dieser Arbeit wurde nachgewiesen, daß es möglich ist, einen kostengünstigen 3D-Scanner für autonome mobile Roboter zu bauen, der diese Anforderungen erfüllt. Der 3D-Scanner erfaßt und vermißt seine Umgebung berührungslos und präzise in nur wenigen Sekunden, ohne daß seine Umwelt mittels passiven Landmarken o.ä. modifiziert werden muß.

Der in dieser Studie vorgestellte 3D-Scanner basiert auf einem 2D-Scanner, wie sie heute schon sehr erfolgreich auf autonomen Robotern eingesetzt werden. Der neue Gesichtspunkt lag in der Möglichkeit, mittels eines handelsüblichen 2D-Laserscanners ohne spezielle PC-Hardware und nur unter Verwendung von Standardschnittstellen das Scannen und Erfassen von kompletten Räumen zu ermöglichen. Dies ist neben Aufgaben in der Robotik wie der Exploration und Navigation auch zur kompletten digitalen 3D-Gebäudemodellierung für integrierte Informationssysteme im Bereich Facility Management/Intelligent Buildings geeignet. Facilities (Liegenschaften) sind komplexe und dynamische Systeme. Ziel ist es, ein digitales 3D-Modell des Gebäudes zu haben, das jederzeit aktuell ist, um beispielsweise die Durchführung und Ausschreibung von Dienstleistungen (Reinigung, etc.) optimal unterstützen zu können. Autonome mobile Roboter können unter Verwendung des in dieser Arbeit vorgestellten innovativen 3D-Sensors kostengünstig die regelmäßige Aktualisierung des digitalen Gebäudemodells unterstützen.

Ein weiterer wichtiger, neuer Aspekt der Arbeit liegt in der Verbindung von Online– und Offline–Algorithmen, wodurch den mobilen Robotern die 3D Daten schnellstmöglich zur Verfügung gestellt und die Rechenzeit optimal genutzt wird. Die in dieser Arbeit erstmalig vorgestellte 3D-Scannerarchitektur ist besonders modular und flexibel. Die Software arbeitet auch mit anderen Hardwarekonfigurationen zusammenarbeiten, beispielsweise mit dem im Abschnitt 2.2.1 vorgestellten Drehmodul.

Die Präzision des realisierten 3D-Sensors hat eine Auflösungsgenauigkeit von 5cm, wobei in jeder Scanebene die Genauigkeit durch die Präzision des Lasers bestimmt wird. Der in dieser Arbeit verwendete Laser hat eine Auflösungsgenauigkeit von etwa 4cm. Die vertikale Präzision wird bestimmt durch die Servogenauigkeit und dem Ansteuerungsverfahren des Servos und liegt bei etwa 0.5°. Für die Ansteuerung des Servos wird das Echtzeitbetriebsystem RT-Linux verwendet. Der 3D-Laserscanner besitzt eine Auflösung von bis zu 113400 Punkten (420 $\times$ 270) bei einem Öffnungswinkel von 150° $\times$ 90°. Damit können Objekte im Nahbereich ab einer Größe von 5cm $\times$ 5cm $\times$ 5cm erkannt werden. Die Zeit für die Aufnahme einer 3D-Szene ist einerseits abhängig von der Scangeschwindigkeit des Lasers und anderseits von der Leistungsfähigkeit der Kommunikationsschnittstelle zur Übertragung der Daten an den angeschlossenen Rechner. In dieser Arbeit wurde lediglich die serielle Schnittstelle zur Datenübertragung verwendet, die in diesem Szenario die Gesamtgeschwindigkeit des 3D-Scanners begrenzt. Die Scangeschwindigkeit für einen kompletten Scan von 150°$\times$ 90° beträgt – je nach Auflösung – zwischen 4 und 12 Sekunden, wobei der Scanner alle 30ms einen 2D-Scan aufnimmt. Die maximale Datenübertragungsgeschwindigkeit (57600 Kbit) der seriellen Schnittstelle begrenzt die effektive Scanrate bei maximaler Auflösung (500 Meßwerte, 12 Bit Auflösung) auf ca 5 - 6 Scans pro Sekunde und erhöht sich bei Reduktion der Auflösung auf beispielsweise 126 Meßwerte entsprechend.

Die Abtastgeschwindigkeit des vorgestellten 3D-Scanners läßt sich durch den Einsatz einer seriellen Schnittstelle des Typs RS422 (max. 500 KBit) nochmals um einen Faktor 4–6 steigern. Weiterhin läßt sich die Abtastgenauigkeit durch einen Update des 2D-Scanners auf einen vermutlich zu Beginn des Jahres 2001 zur Verfügung stehenden neuen Laserscanners nochmals auf bis zu 1cm verbessern.

Das Ergebnisse dieser Arbeit ist ein kostengünstiger, schneller und präziser 3D-Laserscanner, der autonome mobile Roboter in die Lage versetzt, natürliche Umgebungen dreidimensional zu erfassen.

Als nächstes wird der Scanner auf die im Institut verfügbaren mobilen Roboterplattformen (KURT2, Ariadne, vgl. Abbildung 2.2) montiert und das Zusammenspiel des 3D Scanners und der Roboter getestet. Dabei werden insbesondere die noch offenen Probleme der automatischen Überlagerung verschiedener 3D-Szenen und deren Integration in ein präzises 3D-Gebäudemodell untersucht. Weiterhin zu untersuchende Fragestellungen sind eine mögliche Objektklassifikation und die Integration verschiedener Sensor.

Anhang A

Laserscanner

Für die Mitarbeit an dem vorliegenden Kapitel danken wir Adriana Arghir.

Im folgenden werden weitere Details des dieser Arbeit zugrunde liegenden Schmersal Laserscanners wie technische Daten, interne verwendete Algorithmen sowie eine Beschreibung der Protokolle und relevanten Funktions-Schnittstellen.

A.1 Elektrische Schnittstelle

Unter [URLScp] ist eine Beschreibung der Telegrammprotokolle zu finden.

Die elektrische Schnittstelle ist wahlweise implementiert nach EIA RS-422A oder EIA RS-232.

A.2 Übertragungs- und Datenformat

Die Kommunikation erfolgt über die asynchrone serielle Schnittstelle des Sensors. Einstellbar sind folgende Übertragungsraten:

9600 Baud	(Auslieferungszustand)
19200 Baud	
38400 Baud	
57600 Baud	
125000 Baud	
208333 Baud	
312500 Baud	

Ein Datenbyte wird in folgendem Rahmen übertragen:

Der Telegrammaufbau benötigt folgende Erläuterungen, die in Tabelle A.3 zusammengefaßt werden.

Bedingungen

Für die Kommunikation gelten folgende grundsätzliche Bedingungen:

- Der Host ist Master der Kommunikation.
- Eine Anforderung des Hosts unterbricht jede Übertragung des Sensors.
- Ein Kommunikationsabschnitt beginnt immer mit einem Kommando des Hosts an den Sensor. Der Sensor beginnt nie von sich aus eine Übertragung.
- Ein Software-Handshake erfolgt dadurch, daß der Sensor bei Empfang einer übertragungstechnisch (bzgl. Checksumme) korrekten Anforderung mit ACK (06h) antwortet, bei Detektion eines Fehlers hingegen mit NAK (15h).

Start	D0	D1	D2	D3	D4	D5	D6	D7	Stop
Bit	(LSB)								Bit

Tabelle A.1: Datenbyte-Übertragung

STX	ADDR	LENL	LENH	CMD	DATA1	1/4	DATAn	CRCL	CRCH

Tabelle A.2: Telegrammaufbau

Bezeichnung	Datenbreite [Byte]	Erläuterungen
STX	1	Startbyte (02h)
ADDR	1	Host → Sensor: Sensoradresse: 0:Universaladresse 1...255: sensorspezifische Adresse Sensor → Host: Sensoradresse + 80h
LEN	2	Länge der Nutzdaten [Byte], d.h CMD und DATAx; Lower Byte zuerst
CMD	1	Kommandobyte (Telegrammnummer)
DATA1–DATAn	n	optionale Datenbytes (→ CMD)
CRC	2	CRC-16 über das gesamte Telegramm; Lower Byte zuerst

Tabelle A.3: Telegrammaufbau – Erläuterungen

Berechnung der CRC-16

Zur Datensicherung findet eine geschwindigkeitsoptimierte Variante der CRC-16 Verwendung, mit dem Generatorpolynom

$$x^{16} + x^{15} + x^2 + 1. \tag{A.1}$$

A.3 Expansionsalgorithmus

Das Telegramm zur Konfiguration des Schutzfeldes enthält u.a. einen Parameter, der „XOR-Checksumme über die expandierten Schutzfelddaten“ genannt wird. Dabei handelt es sich um einen Zusatzparameter zur Datensicherung der nach einem bestimmten Algorithmus errechnet wird.

Bei der Eingabe als Rechteck werden zwei Grenzwinkel α und β berechnet, wie in der folgenden Abbildung dargestellt.

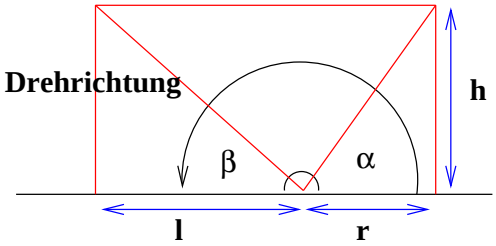


Abbildung A.1: Rechteckige Interpolation

Bei der segmentierten Eingabe wird aus zwei benachbarten Radien zu Segmenteckpunkten die Differenz D berechnet. Durch Division von D durch die Anzahl der im Segment befindlichen Strahlen des expandierten Rasters wird ein Mittelwert des Radiuszuwachses pro Strahl ermittelt. Das Resultat ist eine kreisbogenähnliche Interpolation anstelle einer linearen.

Eine genaue Beschreibung des Algorithmus befindet sich unter [URLScp].

A.4 Beschreibung der Telegramme

Eine genaue Beschreibung der Telegramme befindet sich unter [URLScp].

Von den über 50 zur Verfügung stehenden Telegrammpaare (Frage, Antwort) wurden die im folgenden beschriebenen Telegramme für die Implementierung benötigt.

Telegrammnummer		30h (TGM_REQ_MSRMNT)
Funktionsbezeichner		SndReqMsrmnt(), SndReqMsrmntSeg()
Parameter		
Übertragungsmodus unsigned char	00h	Minimalen Meßwert pro Segment senden ($\rightarrow$ Anzahl der Segmente)
	01h	Alle 501 Meßwerte eines Scans senden
	02h	Minimalen senkrechten Abstand senden. *Nur bei rechteckigem Schutzfeld*
	03h	Eingelernte daten senden, 501 Meßwerte
	04h	Verifikationsdaten senden, 501 Meßwerte
Anzahl der Segmente unsigned int		Gültige Werte: 10, 20, 25, 50, 100, 125, 250, 500
Beschreibung		Es wird festgelegt, in welcher Form der Sensor die Meßwerte an den Host übermittelt

Tabelle A.4: Anforderung Meßwerte

Telegrammnummer		10h (TGM_RST)
Funktionsbezeichner		SndRst()
Parameter		keine
Beschreibung		Software Reset des Sensors. Der Sensor durchläuft eine Initialisierungsroutine wie nach einem Hardwarereset, die konfigurierten Warn- und Schutzfelder bleiben jedoch erhalten. Der Fehlerspeicher wird gelöscht

Tabelle A.5: Software Reset

Telegrammnummer		20h (TGM_SEL_OP_MODE)
Funktionsbezeichner		SndSelOpMode(), SndSelOpModePass()
Parameter		
Modus unsigned char	00h	Konfiguration der Überwachungsbereiche und Parametrierung (nur mit Einrichter-Paßwort)
	01h	Abgleich (nur mit Superuser-Paßwort)
	02h	Rücksetzen auf das Auslieferungspaßwort (benötigt Init-Paßwort). *Nur im Diagnosemodus*
	10h	Diagnose
	20h	Überwachen. Minimale Meßwerte pro Segment werden kontinuierlich ausgegeben

Tabelle A.6: Betriebsmodus wählen

Telegrammnummer		A0h (TGM_RPLY_SEL_OP_MODE)
Funktionsbezeichner		RcvRplySelOpMode()
Parameter		
Status	00h	Moduswechsel erfolgreich durchgeführt
unsigned char	10h	Verarbeitung nicht möglich, da falscher Betriebsmodus (→ Modus 02h)
	12h	Moduswechsel nicht möglich, da falsches Paßwort
	13h	Moduswechsel nicht möglich, da aktives Schutzfeld nicht rechteckig (→ Modi 22h u. 23h)
	80h	Moduswechsel nicht möglich, da Fehler im Sensor
Beschreibung		Der Status gibt an, ob der Betriebsmoduswechsel durchgeführt werden konnte

Tabelle A.7: Antwort Betriebsmodus wählen

Telegrammnummer		66h (TGM_DEF_BR)
Funktionsbezeichner		SndDefBr()
Parameter		
Baudrate	00h	9600 Baud
unsigned char	01h	19200 Baud
	02h	38400 Baud
	03h	57600 Baud
	04h	125000 Baud
	05h	208333 Baud
	06h	312500 Baud
Beschreibung		Sensoreinstellung der Übertragungsrate für die Hostkommunikation (temporär)

Tabelle A.8: Definition Baudrate

Telegrammnummer		E6h (TGM_RPLY_BR)
Funktionsbezeichner		RcvRplyBr()
Parameter		
Status	00h	Einstellung akzeptiert
unsigned char	81h	Interner Fehler (Verarbeitung)
Beschreibung		Der Status gibt an, ob die gewählte Einstellung übernommen wurde

Tabelle A.9: Antwort Definition Baudrate

Telegrammnummer		B0h (TGM_SPLY_MSRMNT)
Funktionsbezeichner		RcvSplyMsrmnt()
Parameter		
Status	00h	Meßwerte werden geliefert
unsigned char	13h	Meßwerte können nicht geliefert werden, da kein rechteckiges Schutzfeld ($\rightarrow$ Übertragungsmodus 02h)
Aktiver Überwachungsbereich unsigned char	00h	Überwachungsbereich 0 aktiv
	01h	Überwachungsbereich 1 aktiv
Anzahl der Meßwerte unsigned int		$\rightarrow$ Anforderung Meßwerte $\rightarrow$ Übertragungsmodus
Meßwerte unsigned int[]		Scan: Bits 0...12:Gemessener Entfernungswert [cm] Bit 13:Gültigkeit (z.B. Blendung) Bit 14:Gesetzt, wenn im aktuellen Scan in diesem Meßpunkt das aktive Warnfeld verletzt wurde Bit 15:Gesetzt, wenn im aktuellen Scan in diesem Meßpunkt das aktive Schutzfeld verletzt wurde Verifikation: Bit 14:Gesetzt, wenn das Schutzfeld in diesem Punkt verifiziert (=verletzt) ist.
Beschreibung		Die Meßwerte eines Scans werden an den Host übergeben. Jedem Meßwert sind außerdem Statusflags zugeordnet, die ggf. Blendung und Schutz- bzw. Warnfeldverletzungen anzeigen

Tabelle A.10: Antwort Anforderung Meßwerte

Telegrammnummer		90h (TGM_RST_ACK)
Funktionsbezeichner		RcvRstAck()
Parameter		keine
Beschreibung		Der Sensor bestätigt das Kommando für den Software Reset und führt ihn durch

Tabelle A.11: Bestätigung Software Reset

Anhang B

Universal Serial Bus

1994 hat eine Allianz von vier Unternehmen (Compaq, Intel, Microsoft und NEC) damit begonnen, den Universal Serial Bus (USB) zu designen. Dem Bus-Design lagen folgende Intentionen zugrunde:

- Verbindung von Computer mit dem Telefon
- Einfach zu benutzen
- Einfach zu erweitern

Im Januar 1996 wurde die erste Spezifikation (1.0) bekanntgegeben. In den heutigen Rechnern finden man die im September 1998 eingeführte Version 1.1. Momentan wird an an der nächsten Generation einer Spezifikation gearbeitet. Es sind allerdings noch keine USB 2.0 Geräte verfügbar.

Der Universal Serial Bus ist strikt hirachisch, die gesamte Kontrolle geht von einem Host aus. Dies bedeutet, daß jegliche Kommunikation von dem Host gestartet werden muß und daß Geräte keine direkte Verbindung zu anderen angeschlossenen Geräten aufbauen können. In der vorliegenden Version erlaubt USB den gleichzeitigen Anschluß von 127 Geräten. Die maximale Bandbreite ist auf 12Mbit pro Sekunde beschränkt. Geräte, die geringe Bandbreite benötigen, wie Mäuse, Tastaturen, Joysticks, etc. kommunizieren mit einer Rate von 1.5MBit/s und nutzen die Fähigkeiten des Busses kaum. Geräte mit hoher Bandbreite wie beispielsweise Audio- und Video-Geräte nutzen etwas 90% des Busses, womit man auf eine tatsächliche Datenrate von 10MBit/s (incl. Protokoll-Overhead) kommt. Desweiteren kann der USB mit bis zu 500mA als Stromquelle den angeschlossenen Geräten zur Verfügung stellen.

Die heutigen Computer weisen an der Rückseite 2 (oder 4) USB Ports auf. An diese Ports können normale Geräte oder Hubs angeschlossen werden. Hubs sind USB-Geräte, die die Anzahl der Ports vergrößern. Die Anzahl der anschließbaren Geräte verringert sich um die Anzahl der angeschlossenen Hubs.

Normalerweise wird der physische Port der Host Controllers (Universal Host Controller (UCHI) von Intel oder Open Host Controller Interface (OHCI) von Compaq) als ein virtueller Root Hub

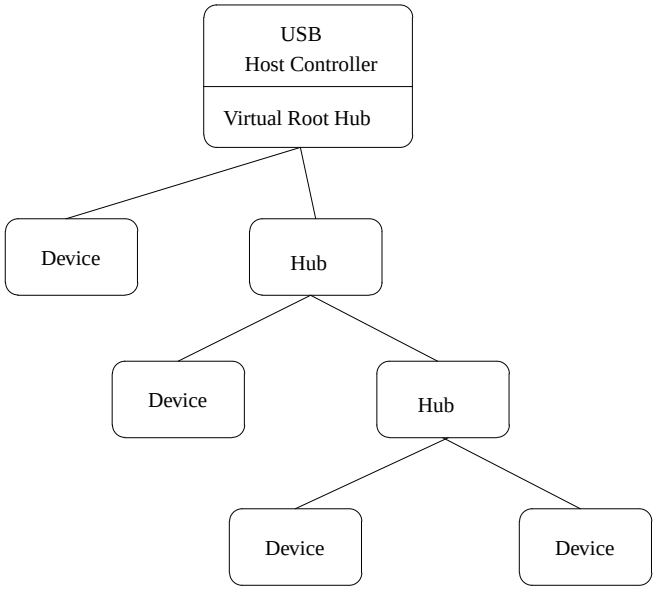


Abbildung B.1: USB Topologie

behandelt, um die Bus Topologie zu vereinfachen. Daher kann jeder Port von dem Linux USB System in gleicher Weise behandelt werden.

Die Kommunikation auf dem Bus geschieht in 2 Richtungen auf vier verschiedene Arten:

Control Transfers werden benutzt, um kleine Datenpackete zu senden oder anzufordern. Dies geschieht, um das Gerät zu konfigurieren. Jedes Gerät muß ein minimale Menge an Kommandos anbieten:

`SET_ADDRESS`, `GET_STATUS`, `GET_INTERFACE`, `SET_INTERFACE`, `GET_DESCRIPTOR`, `SET_DESCRIPTOR`, `SYNC_FRAME`, `SET_FEATURE`, `GET_CONFIGURATION`, `SET_CONFIGURATION`, `CLEAR_FEATURE`.

Bulk Transfers werden benutzt, um Datenpackete mit einer hohen Bandbreite zu senden oder anzufordern. Geräte wie Digitalkameras, Scanner, SCSI Adapter benutzen diesen Transfermodus.

Interrupt Transfer sind ähnlich wie Bulk Transfers, die periodisch gepolled werden. Wenn ein Interrupt Transfer angefordert wird, wiederholt der Host Controller diesen automatisch in einem spezifizierten Intervall (1ms – 255ms).

Isochronous Transfer werden benutzt, um Datenströme mit einer garantieren Bandbreite in Echtzeit zu übertragen. Audio- und Video-Geräte benutzen diesen Transfertyp.

USB verwendet sogenannte Descriptoren, um auf unterschiedlichen Ebenen das Gerät, die Konfiguration, das Interface und die Endpunkte zu beschreiben. Die Descriptoren können abgerufen, und bei den Konfigurationen, Interfaces und Endpunkten können Einstellungen vorgenommen werden (vgl. B.2).

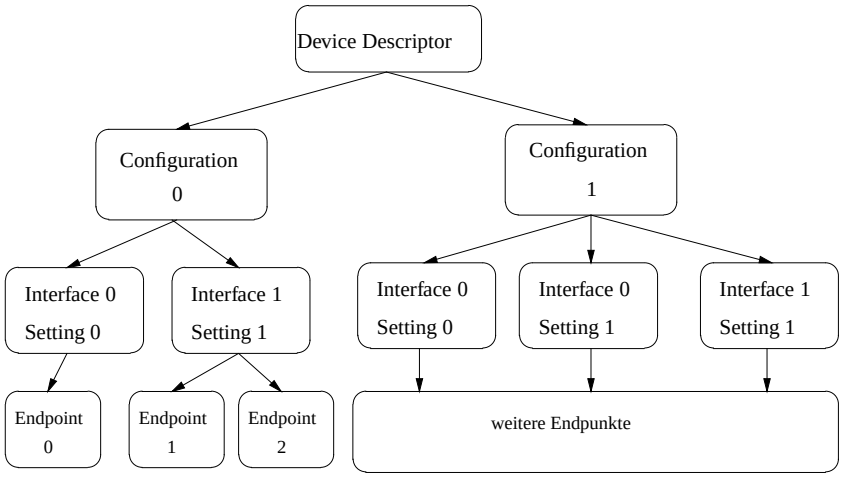


Abbildung B.2: USB Descriptor

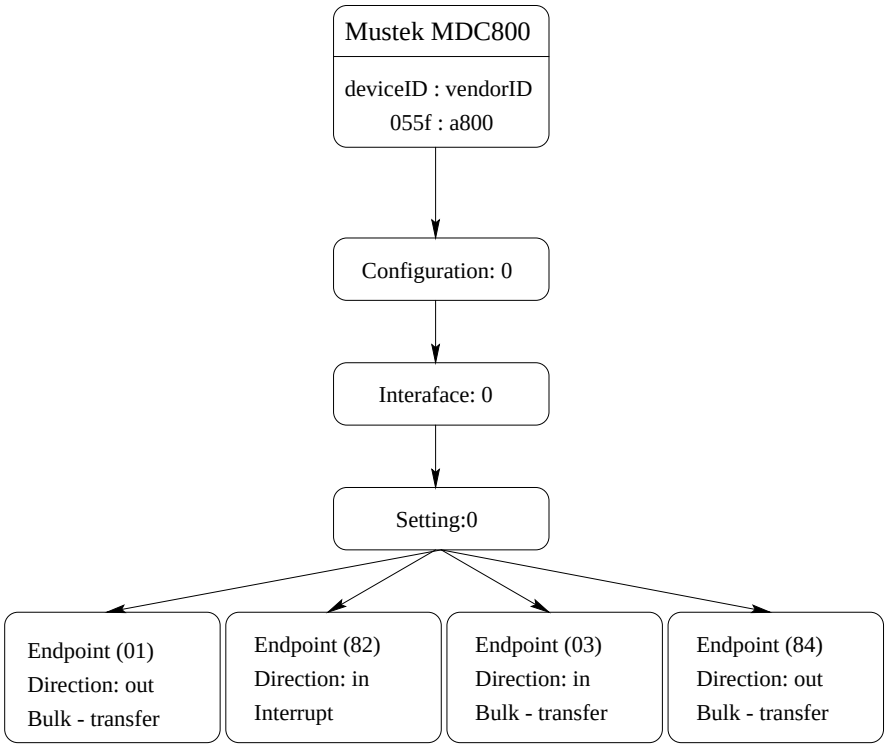


Abbildung B.3: Topologie der MDC800

Anhang C

Die OpenGL–Implementation

Es folgt eine Beschreibung des Anzeigemoduls, Details über dessen Implementation in OpenGL sowie der verwendeten Datenformate, die dem Austasch zwischen Scanner-Applikation und OpenGL dienen.

C.1 Literatur

Die hier benutzte OpenGL-Literatur umfaßt zum einen die OpenGL-Bücher [Cla97] und [Woo97], welche sich bei der Arbeit als besonders nützlich erwiesen haben. Desweiteren fand auch Literatur aus Zeitschriften Verwendung, u.a. eine Einführung in die Programmierung mit OpenGL aus der Zeitschrift *i*X [Mar00], die im Web unter [URLiX] verfügbar ist.

C.2 Zusätzliche Bibliotheken

Zur Verwendung von Hilfsfunktionen, die unter OpenGL nicht direkt zur Verfügung stehen, wurde die Bibliothek *GLUT* von Mark Kilgard verwendet. Im einzelnen wurden folgende Funktionen von uns benutzt:

- Das **Fenster Management** hat die Aufgabe, daß OpenGL Fenster auf die Oberfläche (X11) zu bringen. Zuvor muß das *GLUT* Paket initialisiert werden und verschiedene Parameter gesetzt werden, die direkten Einfluß auf das Verhalten und vor allem die Performance des Programms haben. Diese Funktionen hierfür sind: `glutInit`, `glutInitDisplayMode`, `glutInitWindowSize`, `glutInitWindowPosition` und `glutCreateWindow`
- Der **Anzeige Callback** definiert die Funktion, die Aufgerufen wird, wenn das Fenster neu gezeichnet werden soll. Es kann zum einen ein vom Betriebssystem ausgelöstes Ereignis sein, was das Neuzeichnen des Fensters erforderlich macht, und zum anderen durch das

Programm selbst erforderlich werden (`glutPostRedisplay`). Die Funktion hierfür ist: `glutDisplayFunc`.

- Die **Ereignisschleife** wird als letzter Befehl gestartet (`glutMainLoop`) und kehrt nicht explizit zurück. Zuvor sind die Funktionen `glutSpecialFunc` und `glutSpecialFunc` definiert wurden, die bei Keyboardeingaben aufgerufen werden (vgl. Kapitel 6.3.1).
- Die **Idle-Funktion** (`glutidleFunc`) wird für die Online-Anzeige benötigt. Diese Funktion wird immer dann aufgerufen, wenn das OpenGL-Programm idle ist. Diese Funktion wurde mit der Aktualisierung des Displays verküpft. Damit nicht ständig das Display aktualisiert wird, legt sich dieser Task explizit stets 2 Sekunden schlafen. Dadurch wird erreicht, daß das Bild verhältnismäßig selten aktualisiert wird und es stehen die Ressourcen für andere Programmteile zur Verfügung.

C.3 Funktionen

C.3.1 Transformation und Rotation

Die Quellcodes für Transformation und Rotation sehen wie folgt aus:

```
/* do the modell-transformation */
glTranslatef (X, 0.0 , 0.0);      /* translate X */
glTranslatef (0.0, Y , 0.0);      /* translate Y */
glTranslatef (0.0, 0.0 , Z);      /* translate Z */
glRotatef (rotX, 1.0, 0.0, 0.0);  /* rotate around X-Axis */
glRotatef (rotY, 0.0, 1.0, 0.0);  /* rotate around Y-Axis */
glRotatef (rotZ, 0.0, 0.0, 1.0);  /* rotate around Z-Axis */
```

C.3.2 Die Funktion `DrawQuads`

Hier exemplarisch der Sourcecode der Funktion der Funktion `DrawQuads()`. Die Zeiger `QuadPtr`, `QuadTexturePtr`, `QuadColorPtr` sowie `NumQuad`, die Anzahl der zu zeichnenden Flächen, müssen vorher entsprechend gesetzt werden.

```
void DrawQuads ()
{
   int i = 0, j = 0, k = 0, l = 0;
   i = 0;
   while (i < NumQuad * 12) {
      glBegin (GL_QUADS);
      for (l = 0; l < 4; l++) {
         if (show_texture == 1) {
            glTexCoord2f( (double)QuadTexturePtr[j+1],
```

```
                        (double)QuadTexturePtr[j]);
            j = j + 2;
         } else {
            glColor3f( (double)QuadColorPtr[k++],
                       (double)QuadColorPtr[k++],
                       (double)QuadColorPtr[k++]);
         }
         glVertex3f( (double)QuadPtr[i++],
             (double)QuadPtr[i++],
             (double)QuadPtr[i++]);
      }
      glEnd ();
   }
}
```

C.4 Dateiformate

Die folgenden Beispiele zeigen den prinzipiellen Aufbau von `Points.ogl` und `Data.ogl` an einem konkreten Beispiel. Für eine syntaktische Interpretation siehe Kapitel 6.5.3.

C.4.1 Die Datei `Points.ogl`

```
420
11 29 2 3.75 4.11846
8 30 1 5.3125 8.40997
...
```

C.4.2 Die Datei `Data.ogl`

```
6827 Points
9 27 -2 3.125 3.84159 0.9 0.9 0.9
...
24 Lines
-135 6 -136 -0.307904 0.116042 0.546667 0.1 0.1
-110 6 -138 -0.185688 0.115865 0.54 0.1 0.1
...
21 Surfaces
32 16 -112 0.491071 0.14338 0.1 0.626667 0.1
58 16 -113 0.633296 0.14303 0.1 0.623333 0.1
```

```
60 51 -187 0.513035 0.179337 0.1 0.376667 0.1
39 50 -184 0.444973 0.179064 0.1 0.386667 0.1
...
56 Objects
-77 49 -139 -0.033723 0.201429 0.1 0.1 0.9
-23 49 -139 0.209083 0.201429 0.1 0.1 0.9
-23 280 -139 0.209083 0.661554 0.1 0.1 0.9
-77 280 -139 -0.033723 0.661554 0.1 0.1 0.9
-77 49 -212 -0.033723 0.661554 0.1 0.1 0.9
-23 49 -212 -0.033723 0.661554 0.1 0.1 0.9
...
```

Literaturverzeichnis

[Arr96] Kai O Arras, Sjur J. Vestli, Nadine N Tschichold-Gürman, *Echtzeitfähige Merkmalsextraktion und Situationsinterpretation aus Laserscannerdaten*, Autonome Mobile Systeme, Schmidt G., Freyberger F. (eds.), Seite 57–66, 1996.

[Blo82] Arthur Bloch, *Gesammelte Gründe, warum alles schiefgeht, was schiefgehen kann!*, Goldmann Verlag, 1982.

[Bor98] N. A. Borghese, G. Ferrugno, G. Baroni, S. Ferrari, R. Savare, Autoscan: A Flexible and Portable 3D Scanner, in IEEE Computer Graphics and Applications, Seite 38–41, 1998.

[Con01] Conrad, *Conrad Katalog 2001*, Seite 61, 2001.

[Cla97] Ute Claussen, *Programmieren mit OpenGL*, Springer Verlag, 1997.

[Hab91] Peter Haberäcker, *Digitale Bildverarbeitung*, Hanser Verlag, 1991.

[Her99] J. Hertzberg, E. Rome, F. Kirchner, U. Licht, H. Streich, Th. Christaller, *MAKRO – Bau einer mehrsegmentigen autonomen Kanalroboterplattform*, Zeitschrift *bbr* Wasser und Rohrbau 10, Seite 37–40, 1999.

[Jen99] Patric Jensfelt, Steen Kristensen, *Active Global Localistation for a Mobile Robot Using Multiple Hypothesis Tracking*, in Proceedings of the Workshop on Reasoning with Uncertainty in Robot Navigation, 1999.

[Kuh98] Bernhard Kuhn, *Zeitgenau*, Linux Magazin Verlag, November 1998.

[Mar00] Christian Marten, *OpenGL Tutorial Teil 1–3*, Zeitschrift iX, Dezember 99–Februar 2000.

[McL98] R. A. McLaughlin and M. D. Alder, *The Hough Transform versus The UpWrite*, IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 20 No. 4, Seite 396–400, April 1998.

[Pau98] Michael Pauly, Hartmut Surmann, Marion Finke, and Nang Liang, *Real-time object detection for autonomous robots*, Informatik Aktuell. Autonome Mobile Systeme (AMS 98), 14. Fachgespräch, Springer-Verlag, Seite 57–64, 1998.

[RTL98] RT-Linux Documentation Group, *RT-Linux Manual Project*, 9. Juli 1998

[Seq95] Vitos Sequeira, Joao G.M. Goncalves, M. Isabel Ribeiro, *3D environment modelling using laser range sensing*, Robotics and Autonomous Systems, Seite 81–91, 1995.

[Shi87] Yoshiaki Shirai, *Three-Dimensional Computer Vision*, Springer Verlag, 1987.

[Sur95] Hartmut Surmann, Jörg Huser, and Liliane Peters, *A fuzzy system for indoor mobile robot navigation*, Fourth IEEE International Conference on Fuzzy Systems, FUZZ-IEEE 95, Yokohama, Japan, Seite 83–88, 1995, arwarded with *Robot Intelligence Award*.

[Sur01] Hartmut Surmann and Liliane Peters, *MORIA - A Robot with Fuzzy Controlled Behaviour*, volume 61 of *Studies in Fuzziness and Soft Computing*, chapter Layer Integration, Seite 343–365, ISBN 3-7908-1341-9, Springer-Verlag, 2001.

[Thr00] Sebastian Thrun, Wolfram Burgard, Dieter Fox, *A Real-Time Algorithm for Mobile Robot Mapping With Applications to Multi-Robot and 3D Mapping*, IEEE International Conference on Robotics and Automation, San Francisco, 2000.

[Wal00] Axel Walthelm, Amir Madany Momlouk, *Multisensoric Active Spatial Exploration and Modeling*, Dynamische Perzeption: Workshop der GI-Fachgruppe 1.0.4 Bildverstehen, Ulm, November 2000 / Gregory Baratoff; Heiko Neumann (Hrsg.). - Berlin: AKA Akad. Verl.-Ges., 2000 Seite 141–146.

[Woo97] Mason Woo, Jackie Neider, Tom Davis, *OpenGL Programming Guide*, Addison-Wesley, 1997.

URLs

[URLAmt] Amtec, “amtec product homepage”, `http://www.powercube.de/index-products.html`, September 1999.

[URLCal] Callidus Precision Systems GmbH, `http://www.callidus.de/rahm-dt.htm`, 2000.

[URLFsm] FSMLab, `http://www.fsmlab.com`.

[URLGif] Stuhl-Schnitt, animiertes GIF `http://capehorn.gmd.de:8080/gif/stuhl-schnitt.gif`, April 2000.

[URLGph] GNOME Source Cross Reference, `http://cvs.gnome.org/lxr/source/gphoto/mustek/`, Januar 2000.

[URLHou] XHoughtool 1.1, `http://www.lut.fi/dep/tite/XHoughtool/`.

[URLHp] 3D Laserscanner, `http://capehorn.gmd.de:8080/`, Februar 2001.

[URLiX] iX 12/1999, Graphikprogrammierung, `http://www.heise.de/ix/artikel/1999/12/160`, Dezember 1999.

[URLJ3D] Java 3D(TM) API Home Page, `http://java.sun.com/products/java-media/3d`, Januar 2001.

[URLMes] The Mesa 3D Graphics Library,
http://www.mesa3d.org, November 2000.

[URLMin] Minolta 3D Digitizer,
http://www.3dscanner.ch, Oktober 2000.

[URLOgl] OpenGL - High Performance 2D/3D Graphics,
http://www.opengl.org, Januar 2001.

[URLSch] Schmersal Produkte,
http://www.schmersal.de/d/produkte/n2000/laserlss.html, Februar 1999.

[URLScp] Schmersal Protokoll,
http://capehorn.gmd.de:8080/schmersal/tgmsp114.html.

[URLSie] Siemens Pressemitteilungen,
http://www.siemens.ch/news/presseinfos/pressemitteilungen/005016.htm, Mai 2000.

[URLSic] Sick optic electronic, „PLS: Definition der Telegramme zwischen Benutzerschnittstelle und Sensorsystem über RS-422/RS-232“,
http://www.sickoptic.com/laser.htm, 2000.

[URLThr] Littlejohn Project,
http://www.cs.cmu.edu/afs/cs.cmu.edu/project/mercator/www/littlejohn/, 2000.

[URLVrm] Web3D Consortium,
http://www.vrml.org, Oktober 1999.