Max-Eyth-Schule Kassel
Berufliches Gymnasium

Besondere Lernleistung

Datenverarbeitungstechnik

Thema:
Kartographie eines Raumes durch die mobile Robotereinheit „Robotino“

André Iske

19.01.2010

Inhaltsverzeichnis

Schule:	Max-Eyth-Schule Kassel
Name:	Iske, André
Kurs:	13BG1
Schuljahr:	2009/2010
Leistungskurse:	Datenverarbeitungstechnik / Deutsch

Thema:	Entwicklung eines Steuerungsprogramms zur Kartographie eines Raumes, mithilfe der mobilen Roboterplattform Robotino
Betreuende Lehrkräfte:	Herr John / Herr Dippel

Die vorliegende Arbeit wurde am 24.03.2010 eingereicht.

Ehrenwörtliche Erklärung

"Ich versichere, dass ich die vorgelegte Facharbeit ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Ich bestätige ausdrücklich, Zitate und Quellenangaben mit größter Sorgfalt und Redlichkeit in der vorgeschriebenen Art und Weise kenntlich gemacht zu haben."

Kassel, 24.03.2010 *André Iske*

1. Einleitung

Motivation

Das Thema Robotik hat mich schon seit jeher fasziniert. Die Motivation zur Arbeit mit der mobilen Roboterplattform Robotino, entstand während eines Roboter Projektes unserer Schule im 12. Schuljahr. Damals nahm unsere Schule an einem Roboter Wettbewerb der Firma „Graupner Robotics“[1] teil. Während dieses Projektes bekam ich die Möglichkeit, zur Arbeit mit dem Robotino.

Geschichte der mobilen Robotik

Seit der erste Industrieroboter 1959 seine Arbeit in einem Ford-Werk aufnahm, hat sich die Roboter-Technologie enorm weiterentwickelt. Der ursprüngliche Haupteinsatzort von Robotern waren bestimmte menschliche Arbeitsplätze, an denen sich wiederholende Aufgaben durchgeführt wurden. Diese Arbeitsplätze wurden nach und nach durch Industrieroboter ersetzt.

Durch die enorme Weiterentwicklung im Bereich der Robotik, erweiterte sich das Aufgabenfeld der Roboter mehr und mehr.

Heutzutage sind Roboter aus unserer Industrie nicht mehr wegzudenken. So gut wie alle mechanische Abläufe, die vor 15 Jahren noch manuell durchgeführt wurden, werden heute ausschließlich von Industrierobotern ausgeführt. Viele Produkte erfordern, auf Grund hoher Genauigkeit bei der Herstellung, sogar den Einsatz von Robotern. Andere Produkte wären in Handarbeit viel zu teuer in der Herstellung. Somit bestimmen Roboter längst unseren Alltag.

Neben den festinstallierten Industrierobotern, entwickelte sich eine weitere Art von Robotern, die sogenannten mobilen Roboter Systeme. Mobile Roboter können sich, im Gegensatz zu stationären Industrierobotern, in ihrer Umgebung frei bewegen, und haben somit weitaus vielseitigere Einsatzmöglichkeiten. Beispiele für mobile Roboter-Systeme sind Serviceroboter, wie z.B. Putz- oder Rasenmähroboter, autonome Transportroboter und Roboterfahrzeuge, sowie Unterwasser- und Flugroboter. Zur Zeit versucht man biologisch inspirierte Roboter, wie Kletter- und Laufroboter, menschenähnliche Roboter und Schwarmroboter zu entwickeln.

Eines der größten Probleme bei mobilen Robotern ist die Stromversorgung, da ein Roboter nur solange autonom agieren kann, wie seine Energie reicht. Um dieses Problem zu lösen, versucht man Roboter eigenständig aufladen zu lassen. Diese geschieht entweder durch das selbstständige Aufsuchen einer Ladestation, oder durch die Nutzung von mobilen Energiequellen, wie der Solar-Energie.

[1] Graupner GmbH & Co. KG [25]

Lokalisierung von Roboter-Einheiten

Ein zentrales Problem in der mobilen Robotik ist die Lokalisierung von der Roboter-Einheiten. Zur Weiterentwicklung der mobilen Robotik gibt es zur Zeit mehrere Wettkämpfe, bei denen mobile Roboter Einheiten spielerisch gegeneinander antreten. So ist der RoboCup[2] zum Beispiel ein Fußballwettbewerb, bei dem zwei Teams aus mobilen Roboter-Einheiten gegeneinander Fußball spielen. Die einzelnen Roboter-Einheiten müssen dabei, neben Teamfähigkeit und fußballerischen Anforderungen, auch stets ihrer aktuelle Position und Ausrichtung wissen, in welche Richtung sich das Tor befindet, wo der Ball ist und wo die Gegner sich aufhalten.

Auch im Haushalt, wo Roboter zunehmend an Bedeutung gewinnen, ist die Lokalisierung wichtig. Ein Staubsauge-Roboter erledigt nur dann effektive Arbeit, wenn er weiß, welche Stellen er bereits gereinigt hat.

Da die Lokalisierung des Robotino im Raum auch für die Durchführung meines Projektes unverzichtbar ist, werde ich mich in dieser Hausarbeit sehr viel mit dem Thema der Lokalisierung beschäftigen.

[2] internationalen Forschungs-und Bildungsinitiative - http://www.robocup.org

2. Konzept / Zielsetzung

Ziel meiner Lernleistung ist es, ein Steuerungsprogramm für den Robotino zu entwerfen, welches dem Robotino autonom durch einen Raum navigiert. Dabei soll der Robotino Hindernissen ausweichen und die Wände des Raumes verfolgen. Aufgrund dieser Daten soll eine Art Karte des Raumes und der Hindernisse erstellt werden.

Um dieses Ziel zu erreichen, habe ich über verschiedene Lösungsmöglichkeiten nachgedacht. Hauptproblem des Projektes stellte die genaue Positionsbestimmung sowie die Ausrichtung des Roboters im Raum dar.

Positionsbestimmung durch Odometrie

Mein erster Ansatz, um die Position des Roboters zu bestimmen war die Nutzung der bereits im Roboter installierten Odometrie (siehe Kapitel 3.1.3)**.** Dieser Ansatz musste allerdings verworfen werden, da die Positionsbestimmung durch odometrische Daten zu fehlerbehaftet für eine genaue Kartografie des Raumes war. Durch etwas glatteren Untergrund (z.B. Fliesen oder Laminat) hatten die Räder des Robotino zu viel Schlupf. Auf einem Gummiuntergrund, wie PVC, war eine Fehlerkorrektur durch Gegenrechnen des Schlupfs noch geringfügig möglich. Bei längerer Fahrtzeit allerdings, waren die Positionsdaten auch nach der Gegenrechnung derartig fehlerbehaftet, dass eine genaue Kartografie nicht möglich war. Möglicherweise lässt sich der Fehlerkorrekturalgorithmus noch weiter verbessern. In meinem Projekt habe ich diese Möglichkeit der Positionsbestimmung allerdings verworfen.

Positionsbestimmung durch externen Langstreckensensor

Mein nächster Ansatz zur Positionsbestimmung war die Nutzung eines externen Infrarotabstandssensor. Auch wenn dieser Ansatz, aus später genannten Gründen, verworfen wurde und in meinem jetzigen Stand der Ausarbeitung keine Verwendung findet, möchte ich meine Arbeitsergebnisse an dieser Stelle dokumentieren, da ich bereits viel Arbeitsleistung in die Ausarbeitung dieses Ansatzes gesteckt hatte.

Ziel war es, mithilfe des externen Sensors, einen Großteil des Raumes ab zu scannen, ohne dass der Roboter dabei seine Position ändern muss. Somit sollte die Position des Roboters über die erkannten Wände bestimmt werden.

Der Sensor sollte im Mittelpunkt des Roboters angebracht werden. Zur Erkennung der umliegenden Wände sollte der Roboter eine 360° Drehung durchführen. Während dieser Drehung sollten die Entfernungen des Sensors in einem Feld abgespeichert werden.

Bei der Auswahl des externen Sensors fiel meine Wahl auf den Ultraschall Entfernungsmesser SRF02 der Firma Devantech. Dieser Sensor sollte an die freie serielle Schnittstelle des internen PC104 im Robotino angeschlossen werden. Dazu wurde folgender Schaltplan entwickelt:

Abb. 1

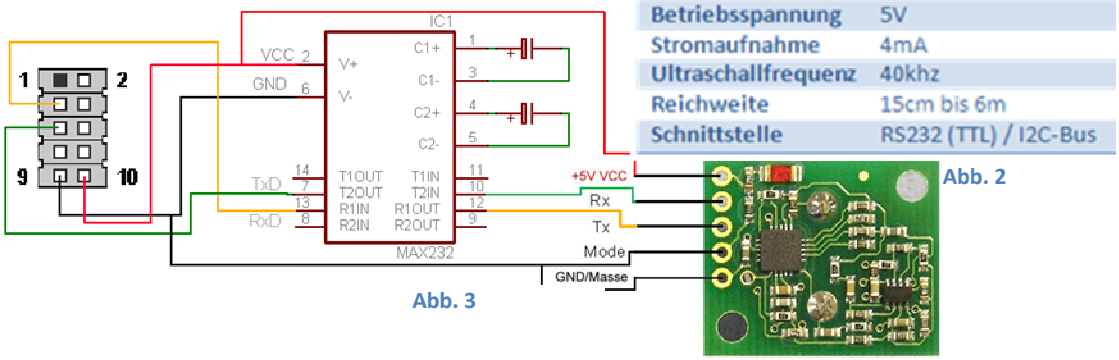


Abb. 3

Betriebsspannung	5V
Stromaufnahme	4mA
Ultraschallfrequenz	40khz
Reichweite	15cm bis 6m
Schnittstelle	RS232 (TTL) / I2C-Bus

Abb. 2

Der PC104 des Robotino kann den von der Sensorelektronik erzeugten seriellen TTL Pegel (0 und 5V) nicht interpretieren, da eine logische 1 bei TTL 5V und bei RS232 -12V entspricht. Deshalb muss das Signal erst durch einen max232 Pegelwandler von TLL auf RS232 gewandelt werden.

Nun musste ein Steuerungsprogramm (Treiber) für den Sensor entwickelt werden, welches dann auf dem Robotino Linux läuft und die serielle Schnittstelle über RTAI anspricht. Der Sensortreiber sollte dann über einen UDP-Stream die Sensor-Werte via WLAN übertragen. An dieser Stelle stieß ich auf mehrere schwer lösbare Probleme: Als erstes benötigte ich das root-Passwort des Linux Systems, um diese Hardwareerweiterung durchzuführen. Da Festo dieses Passwort, aus supporttechnischen Gründen, nur auf Anfrage herausgibt und ich für eine solche langwierige Anfrage keine Zeit hatte, musste ich das Passwort kurzerhand über einen Umweg ändern. Die Durchführung ist im Anhang A.1 beschrieben.

Anders als bei den internen Infrarotsensoren des Robotinos, handelt es sich bei dem SRF02 um einen digitalen Infrarotsensor. Das heißt, die Entfernung wird nicht durchgehend analog zurückgegeben, sondern man muss den Messvorgang über einen bestimmten Befehl starten und anschließend die Messwerte über die serielle Schnittstelle auslesen. Ein Befehl ist immer zwei Byte lang: Das erste Byte beinhaltet die Sensor ID (somit können mehrere Sensoren eingesetzt werden) und das zweite Byte den Befehlscode. Der Befehl 84 (Hex 54) startete Messvorgang in Zentimetern und schickt über den TxD-Pin in einer Endlosschleife die gemessene Entfernung. Die Entfernung wird in 2 Bytes zurückgegeben (aufgeteilt in Low-Byte und High-Byte).

Ein erster Entwurf des Steuerungsprogramms steht bereits (siehe CD). Allerdings gab es größere Probleme bei der Ansteuerung der seriellen Schnittstelle über die RTAI (siehe Kapitel 3.1.2). Aus Zeitgründen suchte ich nach einem anderen Weg, den Sensor auszulesen. Nach intensiver Suche entschied ich mich für einen USB-zu-Seriell-Adapter. Hierfür nutze ich dieselbe Platine, welche auch im optional für Robotino erhältlichen NorthStar Sensors[3] benutzt wird. Die Platine wandelt das TTL RS232 Signal in ein USB-Signal um. Im Linux System erscheint die Platine als neue serielle Schnittstelle (unter /dev/ttyUSB) und kann ohne die RTAI angesprochen werden.

Auf diesem Ansatz hätte ich das Projekt weitergeführt. Allerdings bekam ich glücklicherweise die Möglichkeit, mir einen NorthStar Sensor von meiner Schule für meine Projektarbeit zu leihen. Damit waren die Probleme, bei der Positionsbestimmung des Roboters im Raum, auf einen Schlag gelöst.

[3] optimal erhältliches Navigationssystem für den Robotino (siehe Kapitel 3.2)

Der NorthStar bietet eine sehr präzise Möglichkeit der Lokalisierung des Robotino. Auf die Funktionsweise des NorthStar Navigationssystems gehe ich im Kapitel 3.2. ein. Mein eigentlicher Plan war es, den SRF02 als Langstreckensensor zusätzlich zu den neun internen Infrarotsensoren zu nutzen. Allerdings war es mir zeitlich nicht möglich, ihn bis zur Abgabe dieser Hausarbeit, mit in das Projekt zu integrieren.

3. Grundlagen

Der folgende Abschnitt erklärt die Grundlagen der verwendeten Hardware, die zum Verständnis dieser Arbeit nötig sind.

3.1. Robotino

In diesem Kapitel möchte ich Ihnen das von mir genutzte Robotersystem „Robotino“ der Firma Festo Didactic[4] vorstellen. Bei dem Robotino handelt es sich um ein bereits voll funktionsfähiges, qualitativ sehr hochwertiges, mobiles Robotersystem, das speziell zur Aus- und Weiterbildung entwickelt wurde. Das mobile Robotersystem unterscheidet sich, im Vergleich zum stationären Roboter, durch seine eigene Energieversorgung und Antrieb. Die Energieversorgung des Robotinos ist durch zwei aufladbare 12V Akkus mit 4Ah gewährleistet.

3.1.1. Aufbau / Chassis / Antrieb

Das Chassis des Robotino besteht aus einer lasergeschweißten Edelstahlplattform. Auf das Chassis sind die Akkus, die Sensoren, die Antriebseinheiten und die Kamera montiert.

Ein Großteil der Robotino genutzten Elektronik ist in die Kommandobrücke ausgelagert. Unter der Plastikhaube der Kommandobrücke befinden sich unter anderem der interne Linux PC und der WLAN-Access Point des Robotinos. Die Kommandobrücke ist über eine Stecker-Verbindung mit dem Chassis verbunden.

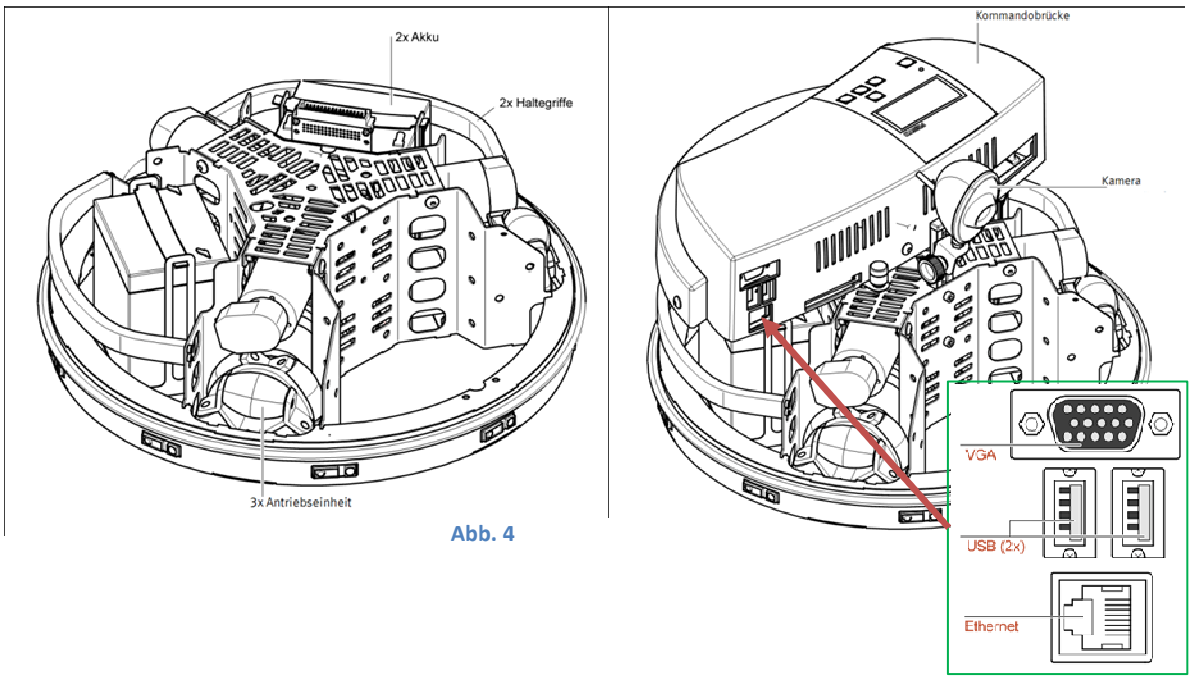


Abb. 4

[4] Eine Tochterfirma von Festo, dem Weltmarktführer in industrieller Weiterbildung

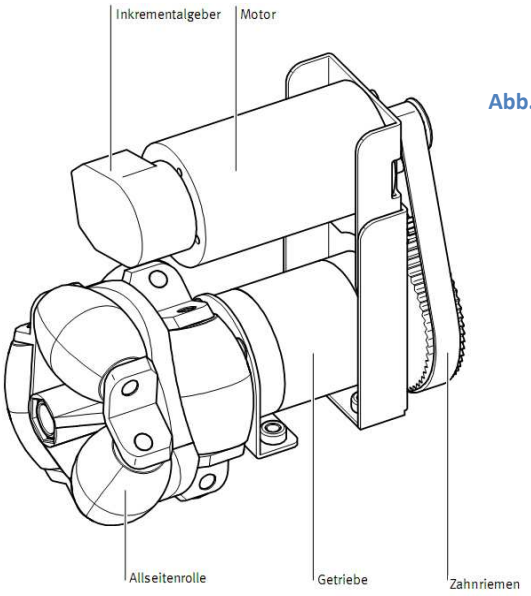


Abb. 5

Abb. 6

Der Robotino verfügt über einen omnidirektionalen[5] Antrieb, bestehend aus drei Antriebseinheiten. Dieser versetzt ihn in die Lage, sich in alle Richtungen zu bewegen und auf der Stelle drehen zu können. Die Antriebseinheiten stehen in einem Winkel von 120° zueinander. Über den Inkrementalgeber oberhalb des Motors kann die reale Motorgeschwindigkeit mit der gewünschten Geschwindigkeit abgeglichen werden und odometrische Daten bestimmt werden. Die Räder des Robotino bestehen aus mehreren voneinander unabhängigen Allseitenrollen. Dadurch kann, durch das Zusammenwirken mit den anderen beiden Antriebseinheiten, eine von der Antriebsrichtung abweichende Bewegungsrichtung erzeugt werden. Für die Gesamtberechnung der Fahrtrichtung, können die Kräfte der Räder auf dem Untergrund vektoriell addiert werden. Dadurch ist der Robotino in der Lage, Kurven zu fahren ohne vorher eine Drehbewegung durchzuführen. Diese Eigenschaft ist gerade bei Echtzeitanwendungen sehr wichtig, da die Zeit für die Drehbewegung gespart wird. Somit ist der Robotino schneller an der vorgegebenen Stelle, als Roboter ohne omnidirektionalen Antrieb.

3.1.2. Interner Linux Embedded PC

Da der Robotino von einem, auf dem Chassis montierten, Linux Embedded[6] PC gesteuert wird, ist es nicht zwingend notwendig ihn mit einem PC zu betreiben. Diverse Demo-Anwendungen können über das LCD-Display gestartet werden.

Abb. 7

Controller-Board.
PC104 MOPSlcdLX
300 MHz / 128 MB SDRAM

Zur Programmierung und zu Diagnosezwecken können auch Monitor und USB-Tastatur an den Embedded PC104 angeschlossen werden. Dies ist allerdings nicht zwingend notwendig, da man sich auch über eine SSH auf dem Robotino Linux System anmelden kann.

[5] In alle Richtungen bewegbar [18]
[6] eingebettetes System [19]

Bei dem Linux System des Roboters handelt es sich um Ubuntu 9.04, das von der Compact Flash Karte abgerufen wird. Das Linux Betriebssystem des Robotinos ist in zwei Schichten unterteilt: Eine reguläre Linux Schicht und eine Real Time Linux (RTAI[7]) Schicht. Der gesamte Hardwarezugriff erfolgt über das RTAI , während der Benutzer durch die API nur Zugriff auf die reguläre Linux Schicht (Userspace[8]) erhält.

RTAI erweitert das Ubuntu-System zu einem Echtzeitbetriebssystem und bietet dem System somit direkten Zugriff auf die Kernel-Ebene. Diese Echtzeiterweiterung ist notwendig, da Sensorabfragen und Motorsteuerungsbefehle extrem zeitkritisch ausgeführt werden müssen.

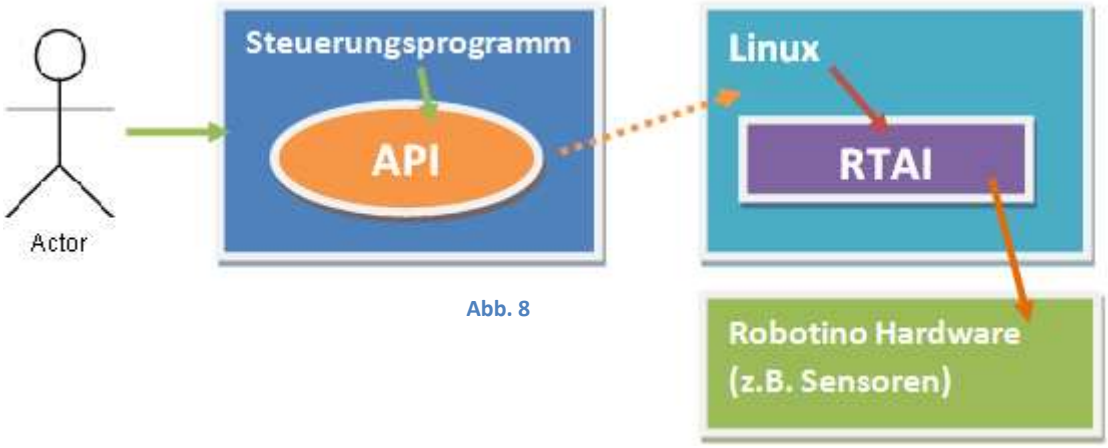


Abb. 8

3.1.3. Sensoren

Der Robotino verfügt über mehrere vorinstallierte Sensoren zum Messen von Entfernungen zu Objekten oder der Bestimmung der aktuellen Motorgeschwindigkeit. Um das Chassis ist ein Kollisionsschutzsensor angebracht, der Berührungen mit Objekten meldet. Des Weiteren besitzt der Robotino ein Kameramodul mit einer maximalen Auflösung von 640x480 (VGA). Die Kamera ermöglicht es ein Live-Bild aufzunehmen und auszuwerten. Somit können Anwendungen, wie Objekterkennung und Verfolgung realisiert werden.

Odometrie[9]

Die Odometrie des Robotinos basiert auf Inkrementalgebern, welche sich oberhalb der Motoren befinden. Mithilfe dieser Inkrementalgeber wird die tatsächliche Drehzahl des jeweiligen Motors bestimmt. Dabei wird die Geschwindigkeit und Entfernung, die ein Rad zurückgelegt hat, gemessen. Mithilfe der Daten aller Motoren, können nun die odometrischen Daten (die Position und Ausrichtung) des Roboters ermittelt werden. Das odometrische Positionsbestimmungsverfahren ist allerdings sehr fehlerbehaftet. Mögliche Fehlerquellen bei der Interpretation der Messung können durch den Schlupf der Räder auf unebenen oder glatten Untergrund entstehen.

[7] Real Time Application Interface [22]
[8] eine Privilegierungsebene über dem Kernel-Modus [23]
[9] griech. Wegmessung []

Infrarot Abstandssensoren

Die wohl wichtigsten Sensoren für die Durchführung meines Projektes sind die neun Infrarot Abstandssensoren[10] des Robotinos. Die Sensoren sind in einem Abstand von 40° um den Roboter verteilt. Somit ist der Roboter in der Lage, zu jeder Zeit sein gesamtes Umfeld nach Objekten und Hindernissen zu untersuchen. Jeder Sensor des Roboters kann über die API einzelnen abgefragt werden und ermöglicht die Abstandsmessung eines Objektes zwischen 4 und 30 cm. Die Auswertelektronik des Sensors misst durchgehend die Entfernung und gibt sie über ein Analogsignal aus. Dieser analoge Wert kann dann über die API abgefragt werden.

$$V = \frac{1}{(R + 0.42)}$$

Verhältnis:
Spannung (V) zu Abstand (R) in cm

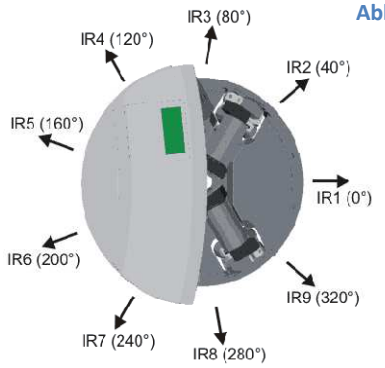


Abb. 9

3.1.4. Programmierung

Die Programmierung des Roboters kann auf verschiedenen Wegen geschehen. Festo Didactic liefert für den Robotino das Programm RobotinoView mit. Dieses Programm richtet sich an Auszubildende, die mit keiner Programmiersprache vertraut sind. Per Drag-and-Drop können hier Funktionsabläufe und ganze Programme „zusammengebastelt werden“. Dabei sendet die Software von einem PC über WLAN Signale an die Motorsteuerung. Weiterhin ermöglicht die Software Sensorwerte anzuzeigen, verändern oder auszuwerten. Dabei werden die Funktionsabläufe im laufenden Betrieb umgesetzt und an den Roboter gesendet.

Da diese Software, im Vergleich zur C++ Programmierung über die API, nur sehr begrenzte Möglichkeiten bietet, habe ich mich in dieser Arbeit nicht weiter mit ihr beschäftigt.

Neben der C++ API, wird in dem Forum www.openrobotino.org noch eine .Net und Java API zur Verfügung gestellt.

Das API ermöglicht den vollen Zugriff auf die Motoren, Sensoren, sowie auf die Kamera des Robotinos. Die Kommunikation zwischen dem Steuerungsprogramm und Robotino wird über TCP und UDP abgewickelt und ist somit vollkommen Netzwerk transparent. Das heißt es spielt keine Rolle, ob das Steuerungsprogramm auf dem Robotino selbst, oder auf einem entfernten Computer ausgeführt wird. Es wird lediglich eine Netzwerkverbindung benötigt.

Alle Klassen des API befinden sich in namespace rec::robotino::com. Zur Vereinfachung der Nutzung kann man diesen namespace als Standard namespace definieren.

[10] Sharp GP2D120 IR Sensor [20][21]

3.2. NorthStar Navigationssystem

Das größte Problem bei der Durchführung meines Vorhabens, war die genaue Lokalisierung des Roboters im Raum. Standardmäßig bietet der Robotino lediglich die Odometrie zur Bestimmung der Position im Raum. Für eine effektive Nutzung, weist die Lokalisierung mithilfe der Odometrie, allerdings zu viele Ungenauigkeiten auf. Daher sind zusätzliche externe Sensoren notwendig.

Abb. 10

Bei dem NorthStar Navigationssystem handelt es sich um ein Infrarot-Ortungssystem, bestehend aus zwei Komponenten: Einem Infrarotsensor und einer Projektionseinheit. Der Sensor besteht aus einem Infrarotsensor, der auf eine Platine befestigt ist, welche zur Signalverarbeitung und Kommunikation mit dem Roboter dient.
Der Sensor wird auf dem Robotino montiert, der Projektor wird zur Decke gerichtet und an einer Wand des Raumes aufgestellt.

Die Ortung wird über zwei, vom Projektor erzeugte, Infrarot-Lichtpunkte durchgeführt. Dieser sendet zwei Lichtmuster, indem er ein Bündel aus parallelen Lichtstrahlen auf eine Fläche, wie beispielsweise die Raumdecke projiziert. Diese zwei Lichtmuster dienen dem Robotino als Orientierungspunkte.

Der auf dem Robotino montierte Infrarotsensor ermittelt die Lage der beiden Lichtpunkte und errechnet anhand diesen Daten seine Position und Orientierung des Roboters. Dafür trägt er die Position der Lichtpunkte in sein internes Sensorkoordinatensystem ein. Die Positionen der Lichtpunkte, werden durch den Einfallswinkel auf den Sensor bestimmt. Die relative Position und Ausrichtung des Roboters innerhalb des Raumes, wird dann über die Position der Lichtpunkte berechnet.

Zur Benutzung von mehreren Projektoren innerhalb eines Raumes, sendet jeder Projektor seine Lichtpunkte durch unterschiedliche Blinkfrequenzen. Am Projektor können neun verschiedene Frequenzen eingestellt werden.

Bevor der Sensor genutzt werden kann, muss er erst auf die jeweilige Deckenhöhe kalibriert werden. Dazu muss der Robotino sich unterhalb der beiden Lichtpunkte befinden.

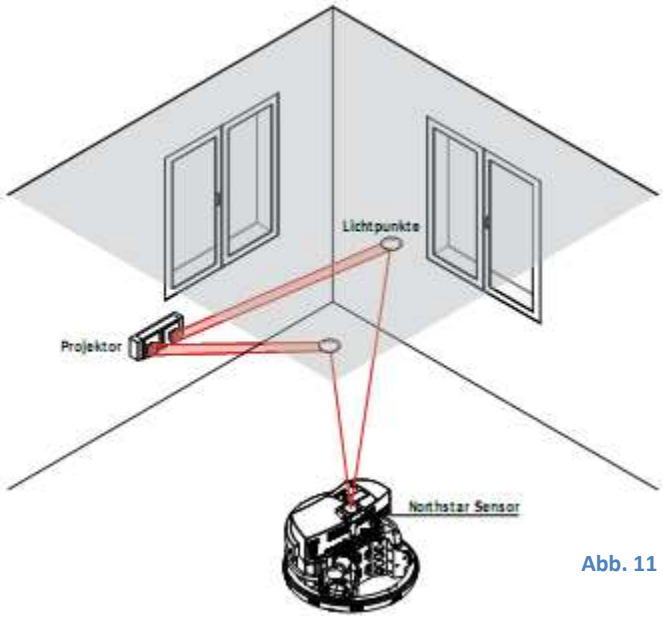


Abb. 11

3.3. Simulationsumgebung RobotinoSim

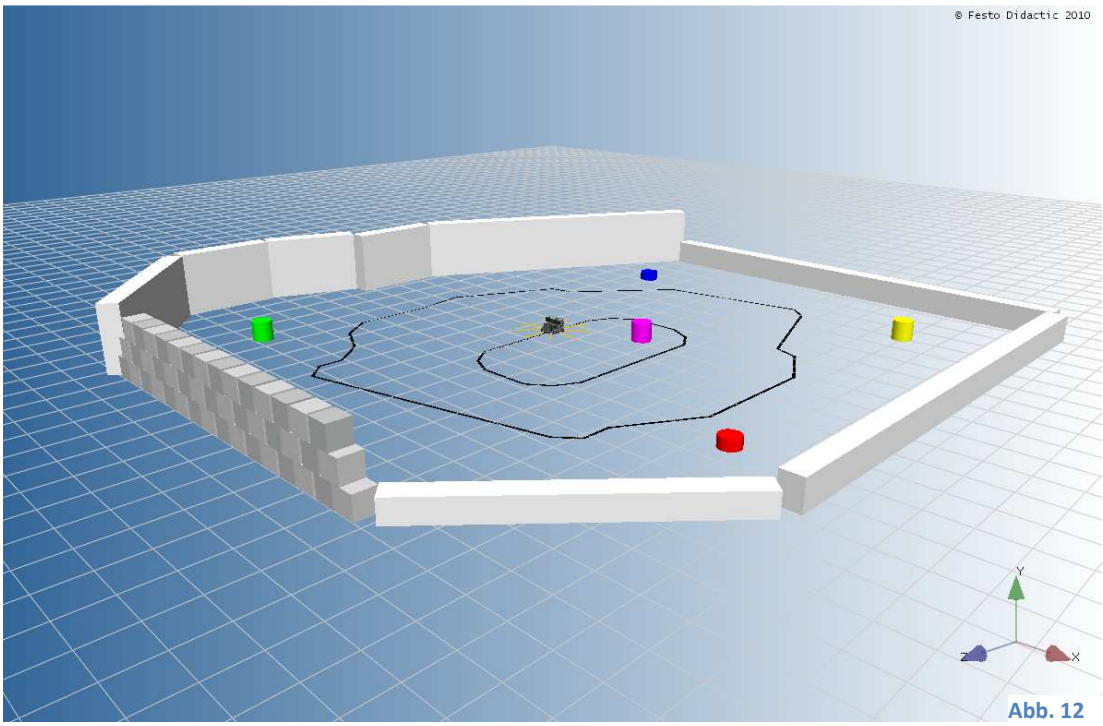


Abb. 12

Bei dem Programm RobotinoSim handelt es sich um eine, von der Firma REC[11] entwickelte, Simulationsumgebung für den Robotino.

Die Anwendung simuliert die Bewegungen des Roboters in einer 3D-Szene und liefert ein „simuliertes“ Kamerabild und die(virtuellen) Messwerte der Sensoren an die API zurück.
Der simulierte Robotino verhält sich dabei nach außen hin so wie der reale Robotino, d.h. für die Steuerung durch das API ergibt sich kein Unterschied. Die Simulationsumgebung startet einen Server auf dem lokalen Rechner (127.0.0.1:8080), der auf die API-Steuerungsbefehle wartet. Möchte man sich nun mit der Simulationsumgebung verbinden, gibt man beim Verbinden lediglich die neue IP-Adresse an.

RobotinoSim simuliert sogar das Vorhandensein eines NorthStar Navigationssystems. Anders als in der Realität muss der NorthStar allerdings nicht mehr kalibriert werden, sondern liefert zu jeder Zeit die hundertprozentig exakte Position. Ebenso wenig gibt es in der simulierten Umgebung den Schlupf der Antriebsräder.

In der kostenlosen Demoversionen bietet RobotinoSim nur eine einzige Standardszene, die nicht verändert werden kann, da die Szene fest in das Programm integriert ist. RobotinoSim Professional bietet zusätzlich die Möglichkeit eigene Szenen zu entwerfen, sowie die Steuerung zusätzlicher Module, wie dem Greifer für den Robotino.

[11] Robotics Equipment Corporation GmbH

4. Durchführung

4.1. RobotinoNavigator

Bei dem RobotinoNavigator handelt es sich um, das von mir entwickelte Programm zu Kartographie eines Raumes mithilfe des Robotinos. Der RobotinoNavigator basiert auf dem MFC SDI Template.

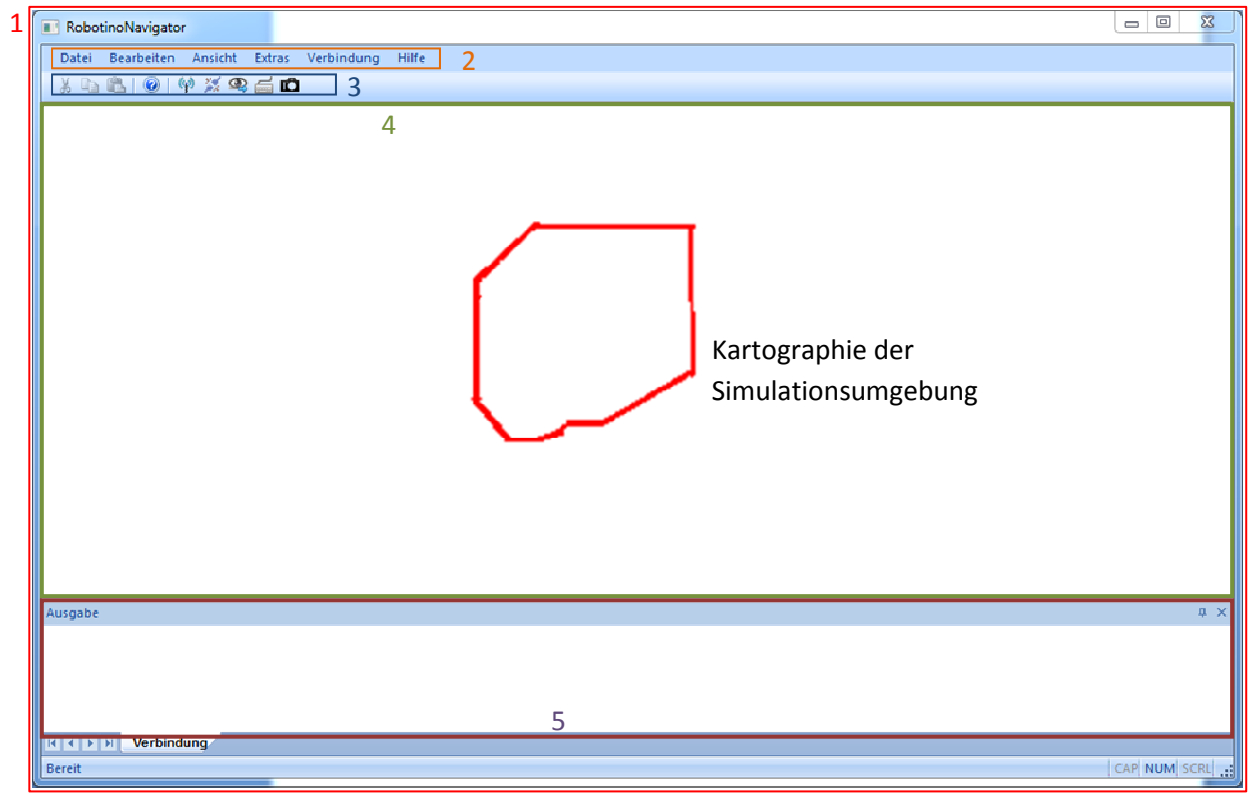


Abb. 13

Aufbau des MFC Fensters

Das Fenster (`MainFrame |1`) des RobotinoNavigator ist in vier Bereiche geteilt. Oberhalb befinden sich eine Menüleiste (`CMFCMenuBar |2`) , sowie eine Symbolleiste (`CMFCToolBar |3`). Da drunter befindet sich der Zeichenbereich (`CChildView |4`) , indem die Umgebung des Raumes eingezeichnet wird. Unterhalb dieses Bereiches ist noch ein Textausgabebereich (`COutputWnd <- COutputList |5`), der den Status der Verbindung zum Roboter ausgibt.

Der Zeichenbereich CChildView

Ein Problem was sich zu anfangs ergab, war das finden einer konvenablen Methode, um das vom Robotino erzeugte Bild in den Zeichenbereich zu zeichnen. Die MFC Anwendung zeichnet den Zeichenbereich in der Methode `OnPaint()` in der Klasse `CChildView`. Die Funktion wird allerdings nicht in einer Endlosschleife aufgerufen, sondern nur, wenn das Fenster den Fokus verliert und anschließend wieder neu bekommt (z.B. minimieren/maximieren).

Mein erster Versuch war es, das Zeichnen in die einzelnen Funktionen auszulagern. Allerdings gab es das Problem, dass beim jeder Aktualisierung des Bildes, die `OnPaint` Funktion erneut aufgerufen wurde und somit der Zeichenbereich zurückgesetzt wurde.

Die beste Lösung für das Problem fand ich, in der Methode, alle Pixel die gezeichnet werden müssen, in einem Bild zu speichern. Das Bild entspricht der Größe des Zeichenbereiches und wird als Member-Variable (`CImage cMapImage`) der Klasse `CChildView` deklariert. In der `OnPaint`-Funktion zeichne ich dann dieses Bild in den Zeichenbereich.

Diese Methode hat den Vorteil , dass ich von jeder Funktion Zugriff auf das Image-Objekt habe und dieses bei Bedarf auch relativ einfach abspeichern kann.

Das Bild wird bei herstellen der Verbindung mit dem Robotino initialisiert. Bei der Initialisierung wird der gesamte Zeichenbereich weiß gezeichnet.

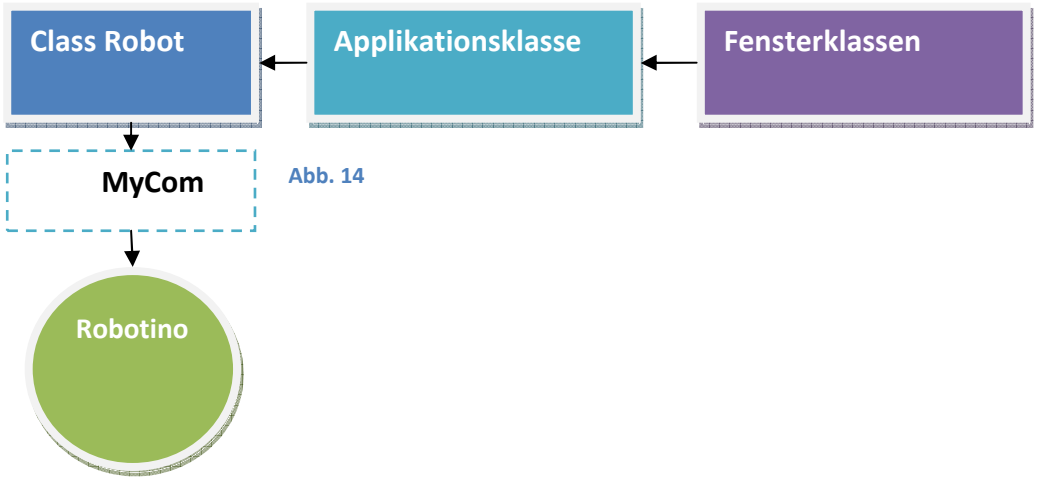


Abb. 14

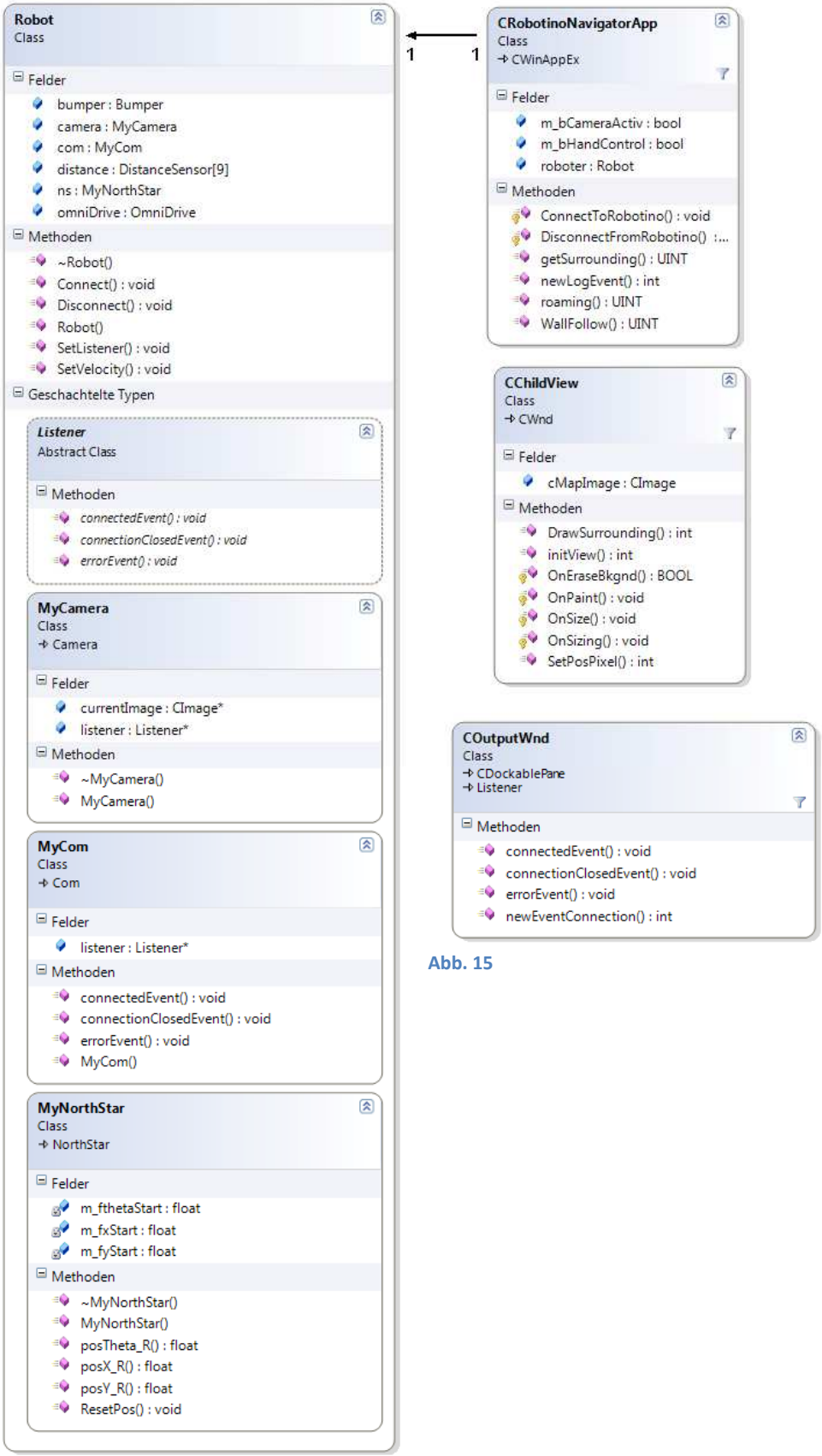

Robot
Class
Felder
bumper : Bumper
camera : MyCamera
com : MyCom
distance : DistanceSensor[9]
ns : MyNorthStar
omniDrive : OmniDrive
Methoden
~Robot()
Connect() : void
Disconnect() : void
Robot()
SetListener() : void
SetVelocity() : void
Geschachtelte Typen
Listener
Abstract Class
Methoden
connectedEvent() : void
connectionClosedEvent() : void
errorEvent() : void
MyCamera
Class
Camera
Felder
currentImage : CImage*
listener : Listener*
Methoden
~MyCamera()
MyCamera()
MyCom
Class
Com
Felder
listener : Listener*
Methoden
connectedEvent() : void
connectionClosedEvent() : void
errorEvent() : void
MyCom()
MyNorthStar
Class
NorthStar
Felder
m_fthetaStart : float
m_fxStart : float
m_fyStart : float
Methoden
~MyNorthStar()
MyNorthStar()
posTheta_R() : float
posX_R() : float
posY_R() : float
ResetPos() : void
1
1
CRobotinoNavigatorApp
Class
CWinAppEx
Felder
m_bCameraActiv : bool
m_bHandControl : bool
roboter : Robot
Methoden
ConnectToRobotino() : void
DisconnectFromRobotino() :...
getSurrounding() : UINT
newLogEvent() : int
roaming() : UINT
WallFollow() : UINT
CChildView
Class
CWnd
Felder
cMapImage : CImage
Methoden
DrawSurrounding() : int
initView() : int
OnEraseBkgnd() : BOOL
OnPaint() : void
OnSize() : void
OnSizing() : void
SetPosPixel() : int
COutputWnd
Class
CDockablePane
Listener
Methoden
connectedEvent() : void
connectionClosedEvent() : void
errorEvent() : void
newEventConnection() : int

Abb. 15

Klassendiagramm RobotinoNavigator

Vereinfachte Darstellung (es wurden lediglich die wichtigsten Klassen, Funktion und Attribute dargestellt)

- Die Kommunikation mit dem Roboter erfolgt in der Klasse Robot.
    - die Klasse Robot verwaltet die Objekte der Akteure des Roboters
        - bumper: Kollisionsstoßleiste
        - camera: Kamera des Roboters
        - com: Objekt zu Netzwerk Kommunikation
        - distance[9]: Feld mit Objekten der Distanz Sensoren
        - ns: NorthStar Sensor
        - omniDrive: omnidirektionaler Antrieb
            - Berechnet aus Richtungsvektor die Geschwindigkeiten für jede Antriebseinheit
        - Klasse Listener
            - Gibt Verbindungsereignisse an Ausgabefenster weiter
        - MyCamera
            - Stellt die Funktionen für das Empfangen von Bildern bereit
        - MyCom
            - Netzwerk Kommunikation mit Robotino
        - MyNorthStar
            - Stellt Funktionen zum NorthStar Sensor bereit
- Die Applikationsklasse der MFC Anwendung besitzt ein Objekt der Kasse Robot.
    - Thread WallFollow : Wandverfolgung
    - Thread Roaming: Raumerkundung
    - Thread getSurrounding: Kartographie des Raums

- Die Klasse CChildView repräsentiert den Zeichenbereich der MFC Anwendung
    - Attribut
        - cMapImage: Karte des Raumes
    - Funktionen
        - DrawSurrounding: zeichnet Entfernung eines Sensors in das Bild
        - SetPosPixel: zeichnet aktuelle Position in das Bild (Wegstrecke markieren)
        - initView: initialisiert das Bild
        - OnPaint: zeichnet Bild in den Zeichenbereich
- Die Klasse COutputWnd stellt den Ausgabebereich unterhalb des Zeichenbereiches dar

4.1.1. Wandverfolgung

Die nachfolgende Dokumentation beschreibt die Funktionsweise der Steuerungsfunktion „WallFollow". Hierbei fährt der Roboter aus seiner Ursprungsposition los, bis er auf eine Wand trifft. Der Wand wird anschließend gefolgt. Der Abstand, den der Roboter zur Wand einhalten soll, wird durch einen DEFINE Wert angegeben.

Da die Auswertungen der Sensoren während der Wandverfolgung sehr zeitkritisch sind, läuft die Wandverfolgung in einem eigenen Thread.

Im Grunde baut meine Wandverfolgungsfunktion auf folgendem Grundgerüst:

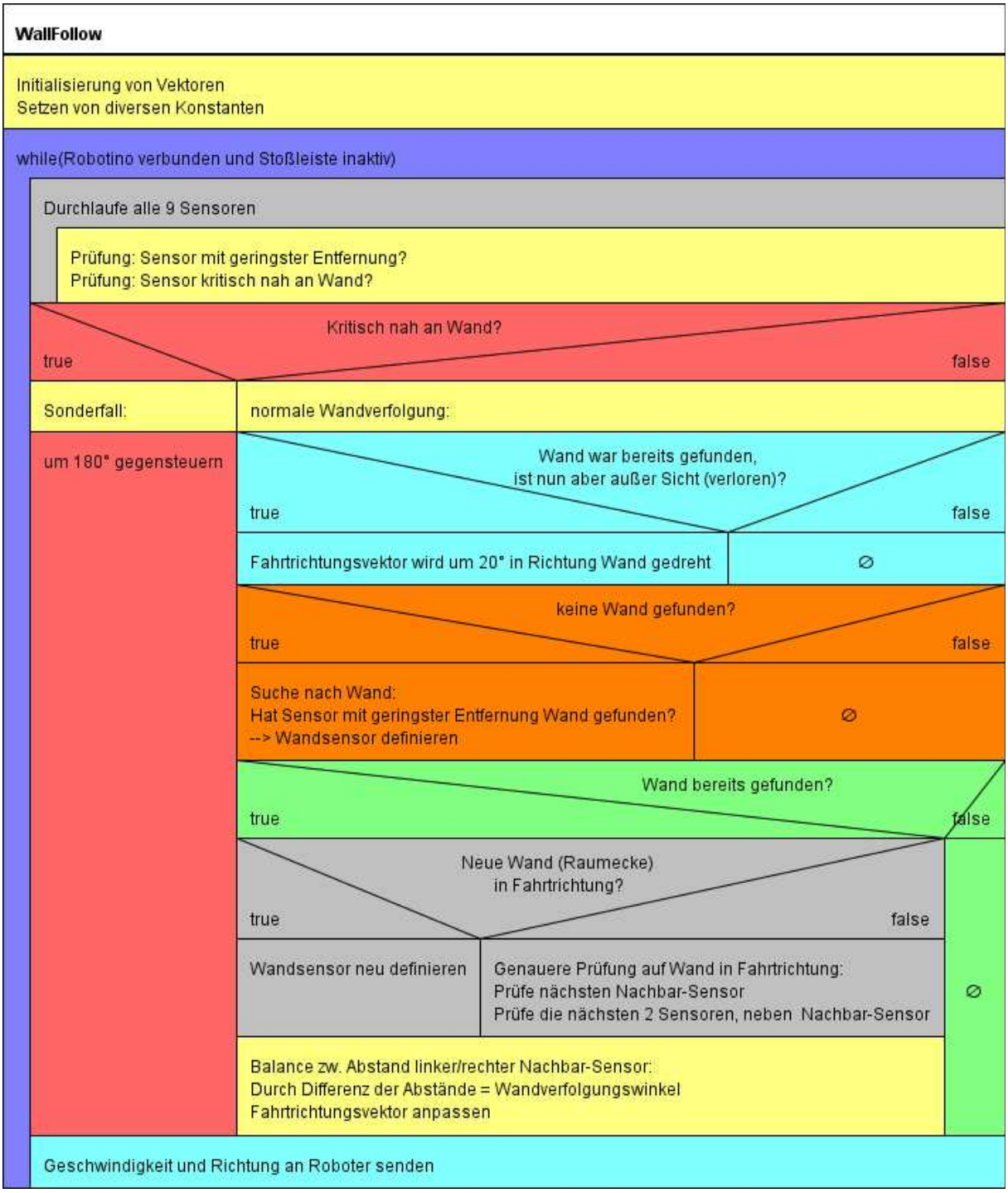


Abb. 16

Da das eben gezeigte Struktogramm den Funktionsablauf nur sehr grob aufzeigen kann, möchte ich im nachfolgenden genauer auf die Wandverfolgungsfunktion eingehen:

Zunächst lege ich mir ein 9×2 Matrix an, in der ich die Richtungsvektoren der einzelnen Sensoren speichere. Dazu lege ich mir zusätzlich den Einheitsvektor $\overrightarrow{ex}$ auf der x-Achse an. Die Infrarot Sensoren des Robotinos sind, wie bereits in Kapitel 3.1.3 beschrieben, in einem Abstand von 40° um den Robotino verteilt. Nun drehe ich den Vektor $\overrightarrow{ex}$ um den Ausrichtungsgrad des jeweiligen Sensors gegen den Uhrzeigersinn, indem ich den Vektor $\overrightarrow{ex}$ mit meiner Rotationsmatrix multipliziere, und speichere dann den daraus resultierenden Richtungsvektor in `sensorDirVector` ab.

```
rotate(ex, sensorDirVector[i],40 * i)
```

$$\overrightarrow{ex} = \begin{pmatrix} 1 \\ 0 \end{pmatrix} \Big| \text{sensorDirVektor} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \overrightarrow{ex}$$

$\text{sensorDirVector}_{9\times 2}$

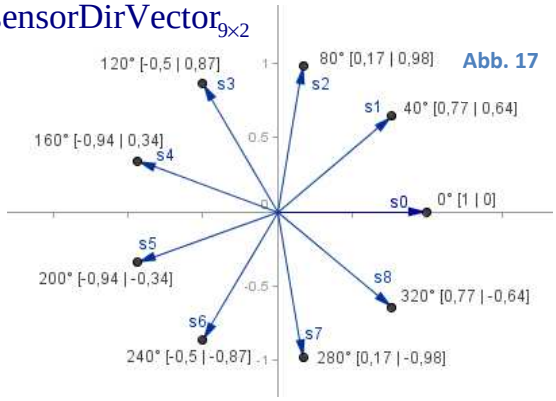


Abb. 17

Um die Fahrtrichtung jederzeit neu setzen zu können, lege ich mir einen Fahrtrichtungsvektor $\overrightarrow{\text{dir}}$ an. In ihm speichere ich die Fahrtrichtung, die ich im Roboter am Ende der while Schleife übergebe.

Nun starte ich die Wandverfolgung in einer Endlosschleife, die nur unterbrochen wird, wenn die Verbindung zum Roboter abbricht, die Sicherheitsstoßleiste durch eine Kollision aktiviert wird, oder der Thread von außen beendet wird.

Ich durchlaufe alle neun Sensoren des Roboters in einer for-Schleife und speichere den Wert und den Index des Sensors mit der geringsten Entfernung (größter Wert) ab. Gleichzeitig prüfe ich an dieser Stelle, ob einer der Sensoren kritisch nah an eine Wand kommt. Sollte das der Fall sein, wird der Richtungsvektor des aktuellen Sensors durch die Entfernung skaliert und in dem Vektor $\overrightarrow{\text{fluchtVektor}}$ gespeichert.

Sonderfall kritisches Gegensteuern

Sollten mehr als ein Sensor diese kritische Nähe erreicht haben, muss der Roboter erst einmal wieder Abstand zur Wand gewinnen. Dazu wird der $\overrightarrow{\text{fluchtVektor}}$, welcher in Richtung Wand zeigt, zuerst normalisiert, dann wird der Gegenvektor gebildet, indem der $\overrightarrow{\text{fluchtVektor}}$ um 180° gedreht wird. Dieser Gegenvektor wird als neuer Fahrtrichtungsvektor $\overrightarrow{\text{dir}}$ gespeichert und sofort an den Roboter gesendet.
In der Simulation ist der Fall, dass der Roboter ein kritisches Gegensteuern durchführen musste, so gut wie nie vorgekommen. Bei der realen Wandverfolgung allerdings, kam es beim Testen immer mal wieder vor, dass der Roboter die Wand minimal berührte und somit die Endlosschleife verließ. Somit wird das kritische Gegensteuern dafür gebraucht, einen sauberen Verfolgungsmechanismus zu ermöglichen und eine Wandkollision in jedem Fall zu vermeiden.

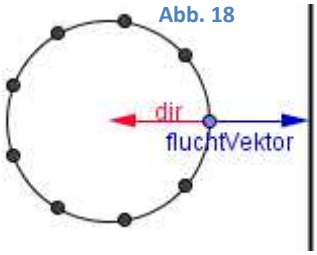


Abb. 18

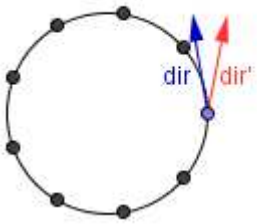
Wenn der Roboter sich nicht kritisch nah an einer Wand befindet, wird die normale Wandverfolgung durchgeführt. Dabei speichere ich den Index des Sensors mit der geringsten Entfernung zur Wand in der Variable `curWallSensor` ab. Dieser Sensor wird also als Wandsensor definiert.

Die **reguläre Wandverfolgung** basiert auf drei Grundabfragen, für die sich jeweils verschiedene Handlungsfälle ergeben:

Wand verloren

Zuerst wird abgefragt, ob `curWallSensor` noch aus dem letzten Schleifendurchgang definiert ist, das heißt eine Wand bereits gefunden wurde, diese aber nun verloren wurde, weil die Entfernung dieses Sensors größer als die festgelegte Konstante ist. Sollte die Wand verloren sein, reagiert Roboter, indem er die Fahrtrichtung um 20° in Richtung Wand ändert. Desweiteren wird die Rotationsgeschwindigkeit, die dem Roboter zusätzlich zum Fahrtrichtungsvektor am Ende der Schleife übergeben wird, in `rotSpeed` gespeichert.

Abb. 19




```
rotateInPlace( dir, -20 );
```

$$\overrightarrow{dir'} = \overrightarrow{dir} \cdot \begin{pmatrix} \cos(20°) & \sin(20°) \\ -\sin(20°) & \cos(20°) \end{pmatrix}$$

Wandsensor nicht belegt , keine Wand gefunden

Die zweite Grundabfrage fragt ab, ob ein Wandsensor momentan definiert ist. Sollte kein Wandsensor definiert sein, wird bei dem Sensor mit der geringsten Entfernung geprüft, ob die Entfernung zur Wand näher, als die vorgebende Konstante ist. Für den Fall, das nun ein Sensor auf eine neue Wand reagiert, wird dieser Sensor als neuer Wandsensor definiert.

Wandsensor vorhanden, der Wand folgen

Die dritte Grundabfrage prüft nun, ob ein Sensor bereits als Wandsensor definiert wurde, d.h. eine Wand zum verfolgen vorhanden ist. Die aktuelle Entfernung zur Wand wird in `wallDist` gespeichert.

Auf neue Wände (Raumecken) in Fahrtrichtung prüfen

Nun ergeben sich eine Reihe von verschachtelten Abfragen, in denen geprüft wird ob sich eine neue Wand in Fahrtrichtung (bzw. ein Knick in der zu verfolgenden Wand) befindet. Zuerst prüfe ich ab, ob ein anderer Sensor näher an der Wand als der aktuell definierte Wandsensor ist und diese Entfernung bereits kleiner als die definierte Konstante `NEUE_WAND_GEFUNDEN_V ist` . Wenn dies zutrifft, dann wird der Wandsensor neu definiert . Die neue Wand ist gefunden und kann verfolgt werden.

Prüfung der Nachbar-Sensoren

Sollte der eben besagte Fall nicht zutreffen, ist zur Sicherheit noch eine genauere Prüfung der Nachbar-Sensoren des aktuellen Wandsensors notwendig:
Als erster wird der direkte Nachbar-Sensor abgefragt und geprüft, ob er näher an der Wand ist als der aktueller Wandsensor. Wenn das der Fall ist, wird der Wandsensor neu gesetzt. Um beim Inkrementieren des Sensor-Index nicht über das Feld, welches die Objekte der Infrarotsensoren verwaltet, hinauszulaufen, ist es notwendig nach dem Inkrementieren mit einem Modulo 9 abzusichern, dass das Feld nach `distance[8]` wieder auf `distance[0]` springt. Somit wird Pufferüberlauf verhindert.

Anschließend werden die zwei Sensoren neben dem Nachbar Sensor, auf eine neue Wand geprüft. Somit bleibt gewährleistet, dass der Robotino auch bei schmalen Raumecken die Wandverfolgung weiterführen kann.

Abb. 20

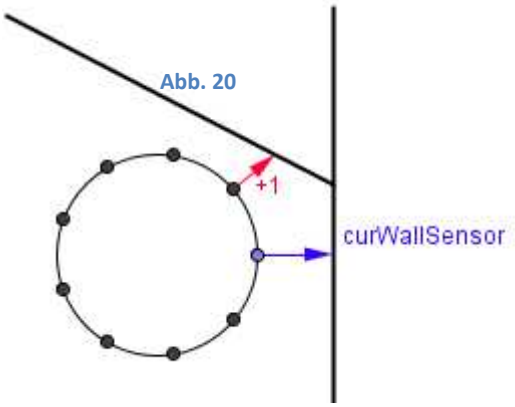
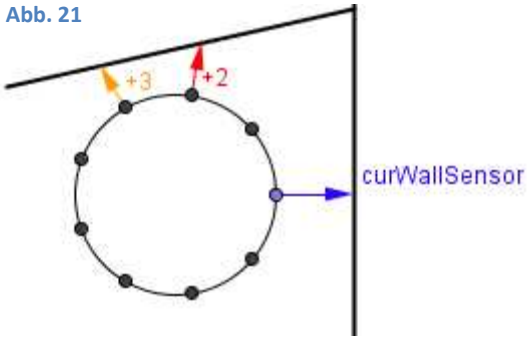


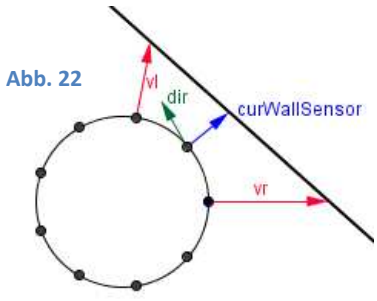
Abb. 21



Fahrtrichtungsvektor festlegen und Wandverfolungswinkel berechnen

Wenn also nun der richtige Wandsensor gefunden und definiert wurde, versucht die Funktion, die Wand möglichst gleichmäßig zu verfolgen. Um dies zu erreichen, versuche ich den Abstand zur Wand zwischen dem linken und rechten Nachbar-Sensor des aktuellen Wandsensors möglichst gleich zu halten. Um diese Balance zu erreichen, bilde ich die Abweichungsdifferenz der beiden Nachbar-Sensoren. Je nach Grad der Abweichung wird dann ein neuer Verfolgungswinkel und damit verbunden auch ein neuer Fahrtrichtungsvektor berechnet.

Abb. 22



Hierbei unterscheide ich zunächst, wie stark die Abweichung ist, indem ich den Betrag der Differenz der beiden Sensoren überprüfe und entscheide dann, welcher Verfolgungswinkel eingeschlagen werden muss. Bei der Auswahl des richtigen Verfolgungswinkels muss außerdem beachtet werden, ob die Differenz positiv oder negativ ist. Ist die Differenz positiv ist der Abstand des rechten Nachbar-Sensors größer. Somit muss auch der Fahrtrichtungsvektor flacher in Richtung Wand zeigen, das heißt, er muss mit einem größeren Gradzahl rotiert werden.

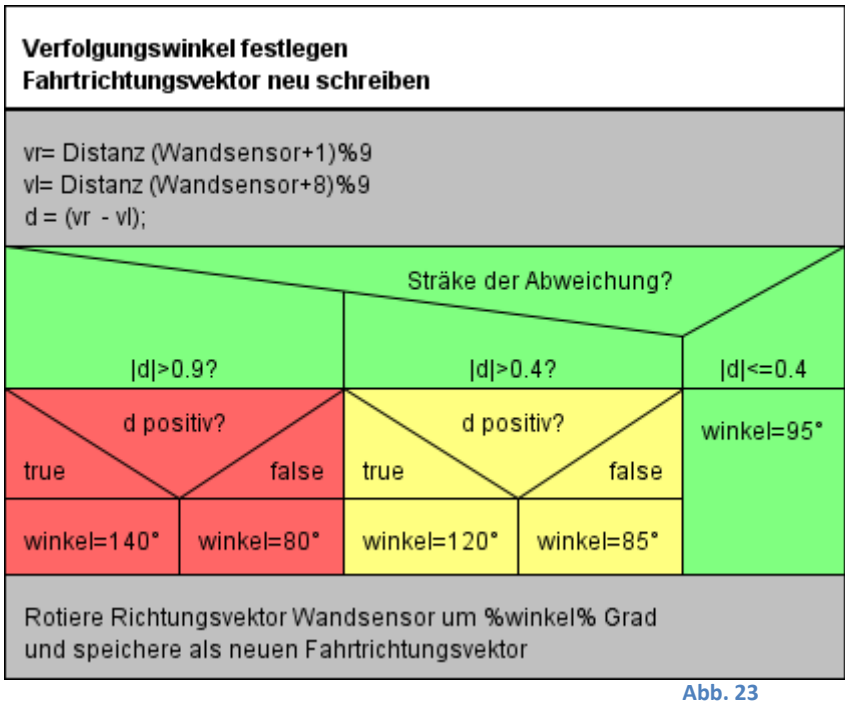


Abb. 23

Geschwindigkeit und Richtung am Roboter senden

Am Ende der Endlosschleife sende ich den neu berechneten Fahrtrichtungsvektor, die Fahrtgeschwindigkeit, sowie die Rotationsgeschwindigkeit über die API-Funktion `setVelocity` an den Roboter gesendet. Abschließend wird auf die neuen Sensorwerte gewartet und somit der Wandverfolgungs-Thread mit dem Kommunikations-Thread synchronisiert.

```
setVelocity( speed * (float)(dir[0]), speed * (float)(dir[1]), rotSpeed );
waitForUpdate();
```

4.1.2. Kartographie des Raumes

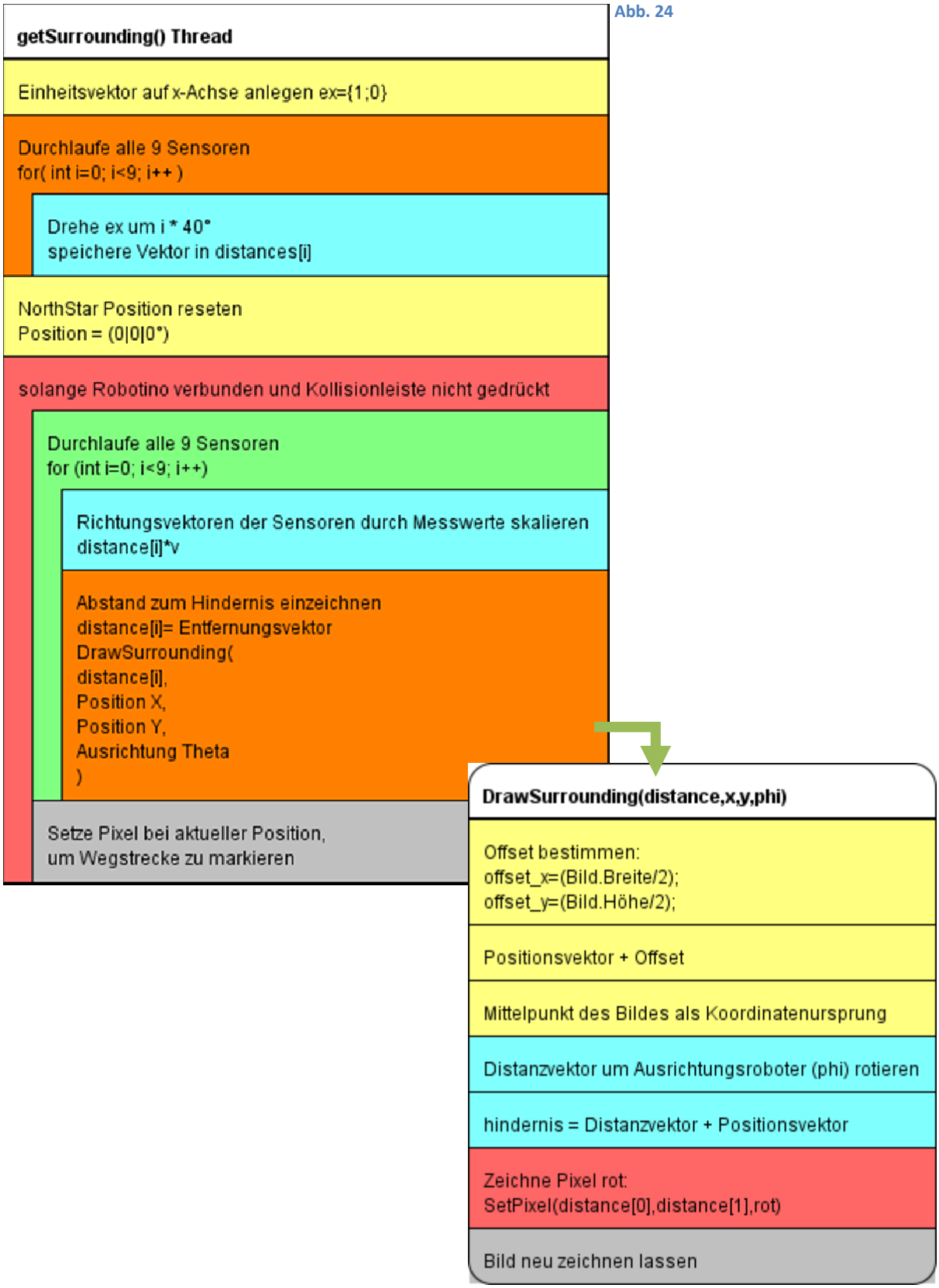
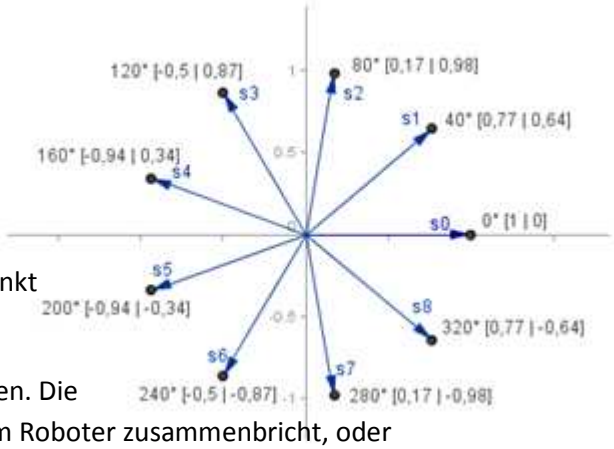


Abb. 24

Entfernung bestimmen

Abb. 25



Das Zeichnen der Umgebung wird durch den Thread `getSurrounding` realisiert. Hierbei werden, wie bei der Wandverfolgung, die Einheits-Richtungsvektoren der neun Sensoren in der 9x2 Matrix `distances` gespeichert.

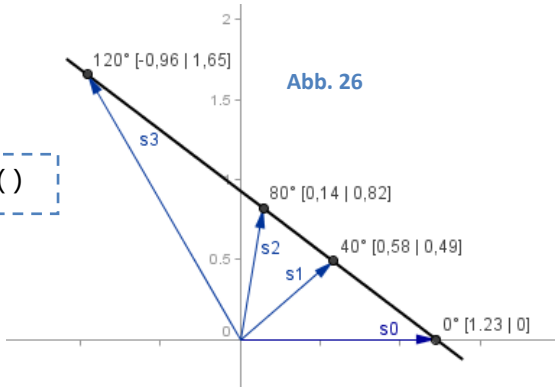
Anschließend wird die Position und Ausrichtung des Roboters auf null gesetzt. Damit wird der aktuelle Standpunkt als Startpunkt definiert.

Nun lasse ich den Thread in einer Endlosschleife durchlaufen. Die Endlosschleife wird nur beendet, wenn die Verbindung zum Roboter zusammenbricht, oder die Kollisionschutzleiste gedrückt wird.

Innerhalb der Endlosschleife werden alle neun Sensoren durchlaufen und die aktuelle Entfernung des jeweiligen Sensors ausgelesen. Mit dieser Entfernung wird dann der Richtungsvektor des Sensors skaliert. Das heißt die Richtung des Sensors bleibt gleich, seine Länge allerdings zeigt die Entfernung zum Hindernis an.

```
scaleVector(tmp,roboter.distance[i].voltage()
```

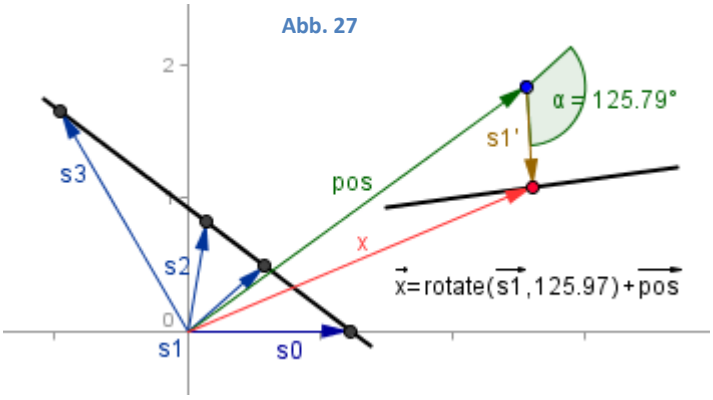
Abb. 26



Entfernung einzeichnen

Dieser skalierte Richtungsvektor, sowie die aktuelle Position und Ausrichtung, werden an die `DrawSurrounding()`-Funktion übergeben. Diese Funktion zeichnet die Entfernung des jeweiligen Abstandsensors auf das Bild.

Abb. 27



Das obige Bild zeigt das Beispiel für das einzeichnen der Wand am Entfernungssensor s1. Der Vektor der aktuellen Position ($\overrightarrow{pos}$), die Ausrichtung des Roboters (hier α), sowie der auf die Entfernung skalierte Richtungsvektor des Sensors ($\overrightarrow{s1}$) wird der `DrawSurrounding()` übergeben.

Die Position der Wand (im Bild als $\vec{x}$ bezeichnet), wird dann über folgende Formel ausgerechnet:

$$\overrightarrow{s1'} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \overrightarrow{s1} \qquad \vec{x} = \overrightarrow{s1'} + \overrightarrow{pos}$$

4.2. Funktionsanalyse und Ausblick

Leider hat mir die Zeit gefehlt, den RobotinoNavigator komplett fertigzustellen. Viele Funktionen sind noch nicht voll ausgereift und funktionieren lediglich in der Simulation. So ist es zum Beispiel notwendig, die die vom NorthStar erhaltenen Positionsdaten durch einen Filter laufen zu lassen, um ein „glatteres“ Ergebnis zu erzielen. In der Simulation muss dies nicht gemacht werden, da hier die Positionsdaten des NorthStar immer exakt die Position im Raum wiedergeben. In der Realität kann eine solch hohe Genauigkeit nicht erzielt werden. Einen solchen Glättungsfilter könnte man durch einen Ringspuffer realisieren. Der Ringspuffer müsste eine gewisse Anzahl von Werten einlesen und dann den Mittelwert der gesammelten Werte ausgeben.

Bis zur Präsentation der Lernleistung, sollen diese Fehler behoben sein, und die fehlenden Funktionen implementiert werden.

A. Anhang

A.1. Ändern des Root-Passwortes

Ab Werk ist das Ubuntu Linux Betriebssystem mit einem von Festo vergebenen Root-Passwort geschützt. Dadurch ergeben sich gewisse Einschränkungen bei der Arbeit mit dem Robotino. Nutzt man lediglich den von Festo zur Verfügung gestellten Funktionsumfang (Robotino API und von Festo bereitgestellte Hardware), reicht es, die Zugangsdaten für den eingeschränkten Nutzer (User: „robotino“, Passwort: „robotino“) zu kennen. Sobald man allerdings tiefgreifende Änderungen, wie das Installieren und Ansteuern neuer Hardware, an dem Robotino durchführen will, ist man zwangsläufig auf das Root-Passwort angewiesen.

Festo gibt das Root-Passwort, aus supporttechnischen Gründen, nur auf Anfrage heraus. Für die, denen eine solche Supportanfrage zu lange dauert, möchte ich hier eine Möglichkeit beschreiben, dass Root-Passwort eigenhändig zu ändern.

Bei einem Linux System werden alle Benutzer und deren Passwörter verschlüsselt in einer Datei abgelegt. Nur der Root-User hat Zugriff auf diese Datei, alle anderen User sind logischerweise per Dateizugriffsrecht „ausgesperrt“. Allerdings werden diese Zugriffsrechte, wie auch in Windows, vom Dateisystem verwaltet. Dieses ist also nur in der Lage, den Zugriff innerhalb des Betriebssystems zu sichern. Um mir nun Zugang zu der Datei zu verschaffen, benötige ich also nichts weiter, als ein anderes Linux System, mit dem ich auf die Festplatte (in diesem Fall CF-Card) zugreifen kann.

Ich habe dazu einen Rechner mit der aktuellen Version von Ubuntu[12] in Verbindung mit einem CF-Kartenleser genutzt. Denkbar wäre hier auch der Einsatz einer virtuellen Maschine.
Nach dem einlegen der Robotino CF-Card sollte man Zugriff auf die Ordnerstruktur des Robotino Betriebssystems haben. Eventuell ist es nötig, einen „mount point“ für die Robotino CF-Card zu setzen.

Da man nun vollen Zugriff auf das Dateisystem des Systems hat, kann man das Passwort in der Datei etc/shadow zurücksetzen. Die Datei enthält den Hashcode der Passwörter aller Benutzer. Diesen Hash können wir zwar nicht entschlüsseln, es reicht allerdings ihn durch einen anderen Hash zu ersetzen. Das heißt wir nehmen den Passwort-Hash des Nutzers robotino und setzen ihn anstelle des Passwortes-Hash für den root-Benutzer. Dadurch ändern wir das Passwort des root-Benutzer auf „robotino“.

[12] Ubuntu Desktop 9.10 32bit - http://www.ubuntu.com/GetUbuntu/download

A.2. CD

Literaturverzeichnis

Alle URLs wurden am 23.03.2010 nochmals auf Richtigkeit geprüft. Nach diesem Datum kann keine Verantwortung mehr für die Verfügbarkeit der Informationen übernommen werden.

[1] Festo Didactic
http://www.festo-didactic.com/de-de/
[2] Diplomarbeit Iordan Pentchev Robotersteuerung in interaktiver Testumgebung
http://opus.haw-hamburg.de/volltexte/2009/809/pdf/diplomarbeit.pdf
[3] openrobotino.org - Robotino Forum
http://forum.openrobotino.org
[4] Robotino Projekt der Hochschule Karlsruhe Fakultät für Informatik
http://code.google.com/p/hska-faki-robotino/
[5] Robotino API Dokumentation
http://doc.openrobotino.org/documentation/OpenRobotinoAPI/1/doc/rec_robotino_com/index.html
[6] Robots in Everyday Human Environments
http://www.control.aau.dk/~tb/wiki/index.php/Robots_in_Everyday_Human_Environments
[7] Projekt: Robotino - Analyse der vorhanden Software
http://www.foo.fh-furtwangen.de/~mueller/ce45_swt/ws07/DokuRobotinoCEB3.pdf
[8] SRF02 Performance
http://knedox.blogspot.com/2009/04/srf02-performance.html
[9] Der I2C-Bus - Was ist das?
http://www.elektronik-magazin.de/page/der-i2c-bus-was-ist-das-21
[10]SRF02 Datenblatt
http://www.robotikhardware.de/download/srf02doku.pdf
[11]MOPSlcdLX
http://emea.kontron.com/products/boards+and+mezzanines/pc104+sbc+and+peripherals/mops+pc104+cpu+modules/mopslcdlx.html
[12]Verfahren zur Erkennung präsentierter Objekte
http://blog.joa-ebert.com/main.pdf
[13]OpenRobotinoApiHowTo
http://doc.openrobotino.org/documentation/OpenRobotinoApiHowTo/OpenRobotinoApiHowTo.pdf
[14]Betreiben des Robotino-Programms ist eine Java3D-Umgebung
https://www.kkrach.de/index.php?id=3,34,000
[15]Robotics Equipment Corporation
http://servicerobotics.eu/index.php?id=3&L=0
[16]OmniWheels - Wissenbasis der Roboternetz Community
http://www.rn-wissen.de/index.php/OmniWheels
[18]Omnidirektional - Logipedia.org.
http://www.logipedia.org/lexikon/omnidirektional
[19] Eingebettetes System – de.wikipdia.org
http://de.wikipedia.org/wiki/Eingebettetes_System
[20]Sharp Distanzsensor
http://www.acroname.com/robotics/parts/R146-GP2D120.html
[21]http://www.acroname.com/robotics/info/articles/irlinear/irlinear.html
[22]RTAI – de.wikipdia.org - http://de.wikipedia.org/wiki/RTAI
[23]Userspace – de.wikipdia.org - http://de.wikipedia.org/wiki/Userspace
[24]Odometrie – de.wikipdia.org - http://de.wikipedia.org/wiki/Odometrie
[25]Graupner Robotics - http://www.graupner-robotics.de/

Abbildungsverzeichnis

Alle Formel wurden mit MathType 6.6 erstellt.

Abbildung	Quelle
1	SRF02 Entfernungssensor - http://www.shop.robotikhardware.de/shop/catalog/product_info.php?products_id=168
2	
3	Schaltplan in Paint gezeichnet
4	Zusammenschnitt aus Bedienungsanleitung des Robotinos
5	Bedienungsanleitung des Robotinos
6	OmniWheels - http://www.rn-wissen.de/index.php/OmniWheels
7	MOPSlcdLX http://emea.kontron.com/_etc/scripts/download/getdownload.php?downloadId=OTg2OQ==
8	Eigenzeichnung mit Word
9	Eigenzeichnung in Paint
10	Bedienungsanleitung NorthStar
11	
12	Bildschirmfoto von RobotinoSim
13	Bildschirmfoto von RobotinoNavigator
14	Eigenzeichnung mit Word
15	Klassendiagramm durch Visual Studio 2008 Klassenassistent + Adobe Photoshop
16	Struktogramm erstellt mit Structorizer | http://structorizer.fisch.lu
17	Eigenzeichnung mit GeoGebra
18	
19	
20	
21	
22	
23	Struktogramm erstellt mit Structorizer | http://structorizer.fisch.lu
24	
25	Eigenzeichnung mit GeoGebra
26	
27	