

# Technische Universität Braunschweig

Teamprojekt

## Koordinierte Ballsuche mit mobilen Robotern

Martin Mikolas, Markus Reschke, Johannes Starosta

Betreuer: René Iser

18. Dezember 2011



Institut für Robotik und Prozessinformatik

Prof. Dr. F. Wahl

## **Erklärung**

Hiermit erklären wir, dass die vorliegende Arbeit selbständig nur unter Verwendung der aufgeführten Hilfsmittel von uns erstellt wurde.

Braunschweig, den 18. Dezember 2011

Unterschrift

## **Zusammenfassung**

Bei der vorliegenden Ausarbeitung handelt es sich um die Dokumentation unseres Teamprojektes zur „Koordinierten Suche mit mobilen Robotern“. Wir werden zunächst die Vorarbeiten vorstellen, auf denen wir aufbauen, bevor wir unser System näher vorstellen. Zum Schluss folgt noch eine kritische Bewertung unserer Ergebnisse.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
<b>2</b>	<b>Softwarearchitektur</b>	<b>3</b>
2.1	Allgemeiner Aufbau . . . . .	3
2.2	Server . . . . .	3
2.2.1	Grafische Benutzeroberfläche (GUI) . . . . .	3
2.2.2	Netzwerkverbindung . . . . .	3
2.2.3	Testclient . . . . .	9
2.2.4	Karte - Wabenkarte und Roboterscan . . . . .	9
2.2.5	Der Regeltakt . . . . .	10
2.2.6	Koordination, Bahnplanung und Kollisionsvermeidung . . . . .	12
2.2.7	Erweiterung von Koordination und Bahnplanung . . . . .	13
2.3	Client . . . . .	15
2.3.1	Ballerkennung . . . . .	15
2.3.2	Sonar-Partikelfilter . . . . .	17
2.3.3	Funktionsweise des Clients . . . . .	18
<b>3</b>	<b>Erstellung der Software</b>	<b>22</b>
3.1	Aufbau des Projektbaums . . . . .	22
3.2	Integration bestehender Projekte in ein neues Buildsystem für den Client . . . . .	23
3.2.1	Warum ein neues Buildsystem? . . . . .	23
3.2.2	Integration der Projekte . . . . .	24
3.2.3	Hinzufügen eines neuen Unterprojektes . . . . .	26
3.3	Erstellung von Client und Server mit Visual Studio 2010 . . . . .	26
<b>4</b>	<b>Nutzung der Software</b>	<b>27</b>
4.1	Benutzung der grafischen Benutzeroberfläche (GUI) des Servers . . . . .	27
4.2	Client . . . . .	29
4.2.1	Kalibrierung der Balldetection . . . . .	29
4.2.2	Einrichtung des Clients . . . . .	33
4.2.3	Benutzung des Clients . . . . .	35
<b>5</b>	<b>Evaluation</b>	<b>37</b>
5.1	Evaluierung des Servers . . . . .	37
5.1.1	Client-Server Kommunikation . . . . .	37
5.1.2	Bahnplanung . . . . .	37
5.1.3	Kollisionsvermeidung . . . . .	37
5.2	Evaluierung des Clients . . . . .	38
5.2.1	Lokalisierung mit den Sonar-Partikelfilter . . . . .	38
5.2.2	Visualisierung der Partikelmengen des Partikelfilters . . . . .	38
5.2.3	False-Positives bei der Ballerkennung . . . . .	40
<b>6</b>	<b>Zusammenfassung</b>	<b>42</b>
	<b>Literaturverzeichnis</b>	<b>43</b>

## Abbildungsverzeichnis

2.1	Server-Client Kommunikationsverbindung. . . . .	4
2.2	Server-Client Datenaustausch. . . . .	6
2.3	Schnittstelle zwischen Server und Client. . . . .	8
2.4	Roboterscanausschnitt. . . . .	10
2.5	Resultierende Wabenkarte. . . . .	10
2.6	Programmablauf des Roboter-Threads. . . . .	11
2.7	Ablauf der Ballerkennung . . . . .	16
2.8	Ablauf der Fahrschleife des Clients . . . . .	20
3.1	Projektbaum nach svn checkout . . . . .	22
3.2	Projektbaum im Unterordner client . . . . .	23
4.1	Pioneer Mobile Robot Server - GUI. . . . .	27
4.2	Robotereinstellungen und -informationen. . . . .	28
4.3	Allgemeine Einstellungen. . . . .	29
4.4	Globale Informationselemente. . . . .	29
4.5	Abstand zwischen Boden und Schachbrett . . . . .	30
4.6	Abfrage der zur Kalibrierung nötigen Parameter . . . . .	31
4.7	Darstellung einer erfolgreichen Kalibrierung der Ballerkennung . . . . .	32
4.8	Empfohlener Aufbau des Laufzeitverzeichnisses . . . . .	34
4.9	Kamerabild bei erfolgreicher Ballerkennung . . . . .	36
5.1	Waben- und Roboterscankarte beim Test der Kollisionserkennung . . . . .	38
5.2	Partikelverteilung im Partikelfilter bei Start nahe der Wände . . . . .	39
5.3	Partikelverteilung im Partikelfilter bei Start in der Raummitte . . . . .	40

## Listings

2.1	Socket Connection - Server. . . . .	4
2.2	Socket Connection - Client. . . . .	5
2.3	Senden und Empfangen. . . . .	6
2.4	Datenstruktur RobotInformation. . . . .	7
2.5	Datenstruktur SearchInformation. . . . .	7
2.6	Datenstruktur PathInformation. . . . .	8
2.7	Thread des Regeltaktes. . . . .	11
2.8	Koordinationstemplate. . . . .	13
2.9	Bahnplantemplate. . . . .	14
2.10	Anmeldung der Klassen in stdafx.h. . . . .	14
2.11	Anmeldung der Klassen in PathManager.cpp. . . . .	14
2.12	Aufbau der config.cfg . . . . .	18
2.13	Ein Beispiel für eine komplette config.cfg . . . . .	18
3.1	CMakeLists.txt für das Buildsystem des Clients . . . . .	24
3.2	CMakeLists.txt für den Sourceordner . . . . .	24
3.3	CMakeLists.txt einer Bibliothek am Beispiel irpCamera . . . . .	25
3.4	CMakeLists.txt des Clients . . . . .	25

# 1 Einleitung

Unsere Aufgabe ist es, eine koordinierte Ballsuche in einem abgegrenzten Raum zu implementieren.

Das Ziel der koordinierten Suche besteht darin, dass die im Raum bzw. im Suchterrain befindlichen mobilen Roboter miteinander das gesuchte Ziel, einen roten Ball, finden sollen. Diese Suche setzt voraus, dass sich die Roboter kollisionsfrei im Raum bewegen. Sie dürfen weder mit Objekten aus der Umgebung, noch mit anderen Robotern, die ebenfalls auf der Suche nach dem gegebenen Objekt sind, kollidieren. Zur Umsetzung dieser Aufgabe standen eine Ad-hoc-Lösung und schließlich eine Server-Client-Lösung zur Auswahl, auf die die Entscheidung fiel. Dies ermöglicht eine parallele Entwicklung von Client und Server, was eine einfachere Aufgabenverteilung im Team ermöglicht. Ausserdem kann dann auf dem Server eine Visualisierung der Ballsuche erfolgen, in dem eine Karte des Raumes, sowie die Position des darin befindlichen Roboter sowie (nach erfolgreicher Suche) die Position des Balls in der GUI des Servers angezeigt wird. Der Client ist dann nur noch für das Abfahren der von Server vorgebenen Positionen, sowie die Lokalisierung im Raum und Ballerkennung in seiner unmittelbaren Umgebung zuständig.

Zum Erreichen des Ziels standen uns mobile Roboter „Pioneer-3DX“ der Firma „adept mobilerobots“ zur Verfügung. Dazu konnten wir diverse Bibliotheken des Instituts nutzen. Die wichtigsten schon vorhandenen Komponenten waren aber der SonarPartikelfilter, sowie die Ballerkennung:

- Der SonarPartikelfilter geht auf ein vorheriges Teamprojekt zurück. Mit ihm konnten wir die Position des Roboters im Raum relativ genau bestimmen. Sie dient als Basis für die Bahnplanung und zum Anfahren der durch diese bestimmten Zielpunkte.
- Die Ballerkennung konnten wir aus der Bachelorarbeit von Tobias Breuer übernehmen. Sie stellte uns eine API zur Verfügung, worüber wir den Ball finden, sowie seine genaue Position bestimmen konnten.

Unsere Softwarearchitektur bestand somit aus folgenden Komponenten:

- Der Server: Er nimmt die Bahnplanung, sowie Kollisionsvermeidung vor und steuert bis zu drei Clients. Gleichzeitig dient er als Visualisierung des aktuellen Status der Ballsuche.
- Der Client: Eine Steuersoftware für die Roboter. Sie ist dafür zuständig, die vom Server vorgegebene Bahnplanung umzusetzen. Dafür nutzt er den Partikelfilter für Ultraschallmessungen, um seine

---

eigene Position im Raum zu bestimmen. Außerdem nimmt er die Ballerkennung vor und bestimmt aus den dabei erhalten Informationen und der durch den Partikelfilter erhaltenen Position die Position des Balles im Raumes.

Im nächsten Abschnitt 2 folgt nun eine genauere Beschreibung unserer Software-Architektur.



## 2 Softwarearchitektur

### 2.1 Allgemeiner Aufbau

Die Software besteht aus zwei Komponenten:

- Der Server, der die Bahnplanung für die Roboter, sowie die Visualisierung der Ballsuche übernimmt.
- Der Client, der die Ballerkennung, sowie Positionsbestimmung vornimmt. Dieser ist ausserdem dafür verantwortlich, den Roboter zu steuern. Dazu lässt der Client den Roboter die durch den Server vorgegebenen Punkte der geplanten Bahn anfahren.

### 2.2 Server

Der Server dient zum einen als zentraler Netzwerk-Einstiegspunkt für alle Clients, und zum anderen auch zur Koordination der Suche und Bahnplanung der verbundenen mobilen Roboter auf der bereitgestellten Karte des Raumes. Die Schnittstelle zwischen Server und Client wurde so gering wie möglich gehalten um sowohl den Server als auch den Client so dynamisch wie möglich verwenden zu können.

#### 2.2.1 Grafische Benutzeroberfläche (GUI)

Zur Visualisierung der im Projekt implementierten Funktionen wurde mittels der objektorientierten Klassenbibliothek der Microsoft Foundation Class (MFC) eine Benutzeroberfläche erstellt. Abgeleitet von der Klasse "CFormView" dient das äußere Fenster als Gerüst für die benötigten Aktions- und Einstellungselemente, sowie für die Darstellung der Karte und des Protokollierungsfensters. Die GUI wird näher im Abschnitt 4.1 ab Seite 27 beschrieben.

#### 2.2.2 Netzwerkverbindung

Für die Kommunikation zwischen Server und Client dient eine TCP-IP-Socket-Verbindung. Wie in Abbildung 2.1 zu sehen, kann nach individuellen Initialisierungsschritten von Server von Client eine Datenverbindung aufgebaut werden, in der die entsprechenden Nutzdaten für die Roboter und für die Serversoftware übertragen wird.

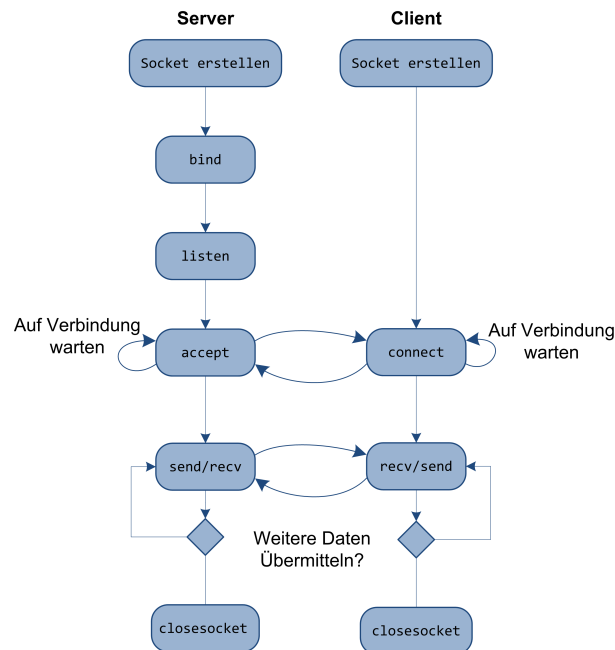


Abbildung 2.1: Server-Client Kommunikationsverbindung.

### Verbindungsaufbau

Zum Erstellen einer Socket-Verbindung muss zunächst ein Socket, unter Angabe von Protokoll und Übertragungsart, erstellt werden. Anschließend wird an diesen Socket die Information über den Service angeheftet. Darunter fallen Kommunikationsart (bsp. TCP oder UDP), Adressdomain (host:port oder UNIX Pathname) und der verwendete Port. Als letzter Initialisierungsschritt, muss die Anzahl der Verbindungen, die der Socket zulässt, angegeben werden. Ist die Initialisierung erfolgt, kann der Socket auf eine Verbindung aus dem Netzwerk warten. Dies geschieht mit der Funktion *accept(...)*, die das Programm bzw. den Thread blockiert, bis eine Verbindung eingegangen wird. Für weitere Verbindungen muss die *accept(..)*-Funktion erneut aufgerufen werden. Der Rückgabewert der Funktion entspricht der Socketnummer und die Variable die als Referenz der Funktion übergebenen wird, enthält nach der eingehenden Verbindung die Informationen über den verbundenen Client. In der Auflistung 2.1 werden die essentiellen Befehle für die Verbindung des Servers zusammengefasst.

```

1 //Zusammenfassung der Implementierung aus der SocketConnection.cpp
2 #pragma comment(lib, "WSOCK32.LIB")
3
4 SOCKET cSocket;
5 sockaddr_in service;
6 int tempsize;
7 struct sockaddr_in tempstr;
8 tempsize = sizeof(tempstr);
9
10 service.sin_family = AF_INET; //Domain
11 service.sin_addr.s_addr = htonl(INADDR_ANY); //initialisierung der Adresse

```

```

12 service.sin_port = htons(7410);           //Port
13
14 //Socket erstellen
15 cSocket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
16 //Service anheften
17 bind(cSocket, (SOCKADDR*)&service, sizeof(service));
18 //Anzahl der Verbindung (CONNECTION_COUNT) angeben
19 listen(cSocket, CONNECTION_COUNT);
20
21 //Auf Verbindung warten
22 int acception = accept(cSocket, (struct sockaddr*)&tempstr, &tempsize);

```

Listing 2.1: Socket Connection - Server.

Um eine Verbindung aufzubauen muss der Client zunächst, wie der Server, einen Socket erstellen. Anschließend kann direkt durch Angabe der Adressdomain, der IP-Adresse und dem Port eine Verbindung mit dem wartenden Server eingegangen werden, wie in Listing 2.2 gezeigt. Während des Versuches eine Verbindung mit dem Server einzugehen wird das Programm bzw. der Thread geblockt bis die Verbindung zustande gekommen ist oder ein interner Timeout den Verbindungsversuch terminiert.

```

1 //Zusammenfassung der Implementierung aus der Connection.cpp das Clients
2
3 #pragma comment(lib, "WSOCK32.LIB")
4 int m_socket;
5 struct sockaddr_in m_serverAddress;
6
7 m_serverAddress.sin_addr.S_un.S_addr = inet_addr('192.168.1.1'); //Server-IP
8 m_serverAddress.sin_port = htons(7410);           //Port
9 m_serverAddress.sin_family = AF_INET;             //Domain
10
11 //Socket erstellen
12 m_socket = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
13 //Mit Server verbinden
14 int check = connect(m_socket, (sockaddr*)&m_serverAddress,
15                     sizeof(m_serverAddress));

```

Listing 2.2: Socket Connection - Client.

## Datenaustausch

Zum Senden und Empfangen von Daten wurde eine Reihenfolge von Aktionen festgelegt, da ein Paket, das gesendet wurde, auch von der Gegenseite empfangen werden muss. Dieser Ablauf wird in Abbildung 2.2 dargestellt. Neben den Nutzdaten für und über die mobilen Roboter werden desweiteren ACK's gesendet, die ein erfolgreiches Empfangen signalisieren. Diese Implementierung wurde zunächst für die ersten Testreihen genutzt und sollte im späteren Verlauf entfernt werden. Jedoch erwiesen diese sich als nützlich zur Fehlererkennung bei einer Übergabe falscher Daten und sind aus Programmstabilitätsgründen erhalten geblieben.

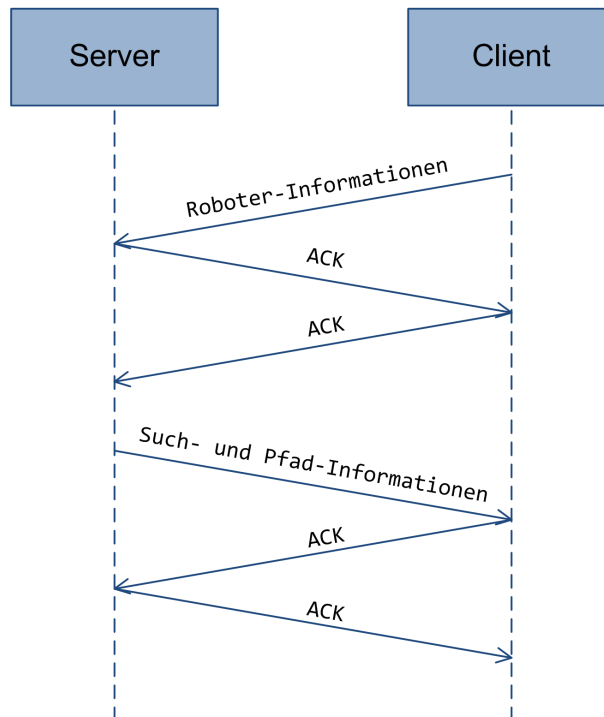


Abbildung 2.2: Server-Client Datenaustausch.

Der Austausch der Informationen erfolgt in erstellten Datenstrukturen, die in Kapitel 2.2.2 erläutert werden. Diese angelegten und mit Informationen erfüllten Strukturen werden zu einem Zeitpunkt vom Server bzw. Client gesendet und vom jeweils anderen Kommunikationspartner empfangen. Dies erfolgt über die in Listing 2.3 angegebenen Funktionen. Mittels der Übergabe des Sockets der Verbindung werden die Daten zwischen den korrekten Kommunikationspartnern ausgetauscht. Sowohl die zu sendenden Daten als auch die zu empfangenden Daten befinden sich in der Struktur, die als Referenz an die Funktion übergeben wird. Nach dem Ausführen der Funktion wird ein ganzzahliger Wert zurückgegeben. Dieser entspricht der Anzahl der übermittelten Bytes. Sollten keine Daten übertragen werden gibt die Funktion einen negativen Wert zurück.

```
1 int check;
2 struct Information info;
3
4 //Daten senden (die Variable muss Daten enthalten)
5 check = send(getSocket(),(char*)info,sizeof(Information),0);
6
7 //Daten empfangen
8 check = recv(getSocket(),(char*)info,sizeof(Information),0);
```

Listing 2.3: Senden und Empfangen.

## Datenstrukturen für den Datenaustausch

Die Datenstrukturen für den Datenaustausch konzentrieren sich auf die wesentlichen Informationen, die zwischen dem Server und einem Client ausgetauscht werden müssen. Ziel war es eine möglichst kleine und allgemeine Schnittstelle zu konstruieren um eine Verwendung mit weiteren Robotern nicht auszuschließen. Beispielsweise wäre die Übertragung der Ultraschallsensordaten des Roboters, nicht durch Roboter anderer Modelle möglich und wurde daher auch nicht in die Übertragungsstruktur aufgenommen. Die in Listing 2.4, 2.5 und 2.6 zu sehenden Strukturen, sind die Informationsträger, die zwischen dem Server und dem Client übertragen werden.

```
1 struct RobotInformation
2 {
3     Point pos;           //Aktuelle Standardsinformationen des Roboters
4     int angle;           //Angle in Grad
5
6     bool ballDetected;   //Ball gefunden?
7     Point ballPos;       //Position des gefundenen Balles
8
9     bool needPathTable;  //Neuen Pfad anfordern
10
11     bool pointReached;   //Ein Punkt wurde erreicht
12 };
```

Listing 2.4: Datenstruktur RobotInformation.

Die RobotInformation-Struktur aus Listing 2.4 beinhaltet Informationen über den aktuellen Status des Roboters. Dazu gehören die Koordinaten an denen sich der mobile Roboter zum Übertragungszeitpunkt befindet. Zur Visualisierung auf der Karte des Servers wird außerdem noch die Ausrichtung des Roboters übermittelt, die als Winkel der x-Achse und der Blickrichtung definiert ist. Sollte der Roboter einen Ball im Raum detektieren, wird diese Information inklusive der Ballposition übertragen. Die Ballposition ist dabei für die Darstellung auf der Karte nötig.

Während der Roboter die vom Server generierten Wege abfährt, werden zwei Signalisierungstypen benötigt. Zum einen wird, für die Darstellung der Karte des Servers, signalisiert, dass ein Punkt der Liste erreicht wurde, der beim nächsten Zeichnen der Karte gelöscht werden kann und zum anderen wird signalisiert, dass ein neuer Pfad benötigt wird, sobald alle Punkte abgearbeitet wurden.

```
1 struct SearchInformation
2 {
3     int transmissionPathTableSize; //Pfadlaenge von PathInformation
4
5     bool ballDetected;             //Ball wurde gefunden
6
7     bool pause;                    //Suche Aktiviert/Deaktiviert
8 };
```

Listing 2.5: Datenstruktur SearchInformation.

In der Struktur `SearchInformation` aus Listing 2.5 werden Informationen vom Server an den mobilen Client gesendet. Diese Informationen signalisieren dem Roboter den Zustand der Suche. Pausiert die Suche, so sollte der Roboter sich nicht im Raum bewegen. Andernfalls läuft die Suche und der Roboter darf die Fahrt aufnehmen. Wurde der Ball von einem der in der Suche verwendeten Roboter gefunden, ist die Suche beendet und damit können auch alle Roboter ihre Suche beenden. Dies wird durch die Variable `ballDetected` signalisiert. Desweiteren muss die Größe des zu übertragenden Pfades mitgeteilt werden, denn zur Übertragung von Daten muss stets die Größe der zu übermittelnden Daten angegeben werden.

```
1 struct PathInformation
2 {
3     std::vector<Point> pathList;    //Liste des Pfades
4 };
```

Listing 2.6: Datenstruktur `PathInformation`.

Die `PathInformation` aus Listing 2.6 übermittelt die vom Server generierten Pfade, die durch den Bahnplaner entstanden sind. Dieser Vorgang erfolgt, sobald der Client eine neue Liste von Wegpunkten anfordert.

Aus den Strukturen der Listings 2.4, 2.5 und 2.6 ergibt sich schließlich die in Abbildung 2.3 dargestellte Schnittstelle, die im definierten Regeltakt zwischen Server und Client übertragen wird.

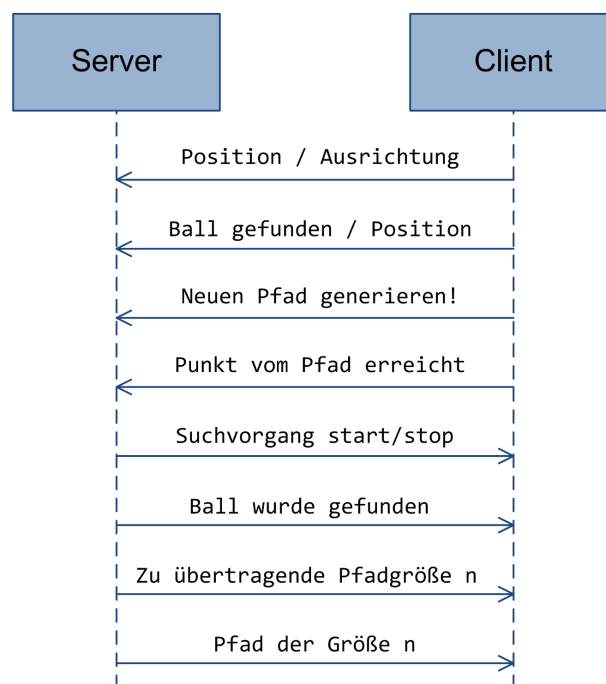


Abbildung 2.3: Schnittstelle zwischen Server und Client.

### 2.2.3 Testclient

Zum Testen der Anwendung wurde eine Testclientumgebung implementiert. Diese dient zur Prüfung der Netzwerkverbindung und Simulation der mobilen Roboter ohne die Verwendung eines realen Roboters. Die Client-Konsolen-Applikation verbindet sich nach dem Starten mit dem Server und füllt die Daten der Schnittstelle mit fiktiven Informationen. Um das Verhalten eines Roboters zu simulieren, wird, bei einer gestarteten Suche, in jedem Zeitschritt die benötigte Orientierung ermittelt und in diese Richtung eine vordefinierte Länge zurückgelegt. Kollisionen und anderen unerwarteten Vorkommnisse werden nicht berücksichtigt.

### 2.2.4 Karte - Wabenkarte und Roboterscan

Zur Navigation der Roboter muss der Raum, in dem sie sich bewegen sollen, bekannt sein. Dazu sind in einer Karte Positionen von Hindernissen (z.B. Wände) und freie Flächen hinterlegt. Die Punkte und die dazugehörigen Informationen sind in einer Bitmap-Bilddatei kodiert. Sie besitzt eine Grauwerttiefe von 8 Bit und kann damit 256 verschiedene Informationen über einen Punkt enthalten. Die Datei, die zur aktuellen Beschaffenheit des Raumes, passt muss in das System geladen werden und in einem  $n \times m$  Array gespeichert werden, um weitere Berechnungen im Programm durchzuführen. Dabei entspricht  $m$  der Höhe und  $n$  der Breite des Bildes in Pixel. Für den Roboter bedeutet dies, dass dieser sich nur auf den Koordinaten im Raum bewegen darf, die auf der Karte einen Wert von 255 besitzen. Alle anderen Werte bzw. Punkte sind nicht befahrbar, da sie ein Hindernis darstellen.

Da sich schnell herausstellt, dass die Berechnung auf einer Karte mit einer Höhe und Breite von jeweils 800 Pixeln mindestens 640.000 Rechenschritte benötigt und dadurch das Berechnen eines Pfades viel Zeit in Anspruch nimmt, kam die Idee den Vorgang zu beschleunigen.

Die Lösung ist die Karte zu verkleinern, da eine so hohe Auflösung des Raumes nicht benötigt wird. Dazu wird eine neue Karte erstellt, die in der Höhe und Breite um den Faktor  $F$  kleiner ist. Damit wird die Rechenzeit um den Faktor  $\frac{1}{F^2}$  beschleunigt. Die Pixel der neuen Karte werden im Programm als Wabe (in Anlehnung zu Bienenwabe, engl. „comb“) und die Karte als Wabenkarte bezeichnet. Eine Wabe repräsentiert die Information von  $F^2$  Pixeln. Um eine maximale Sicherheit zu erhalten, ist eine Wabe für den Roboter nur befahrbar, wenn alle korrespondierenden Pixel der Originalen Karte für den Roboter befahrbar sind. Das stark vergrößerte Ergebnis der Umwandlung einer Roboterscankarte (Abbildung 2.4) zu einer Wabenkarte wird in Abbildung 2.5 dargestellt, wobei die grünen Kästen den Rand der Wabe markieren und in der Datenstruktur im System nicht vorhanden sind. Die GUI ermöglicht, wie in Kapitel 4.1 beschrieben, das Wechseln der Ansichten für die Darstellung der Karte.

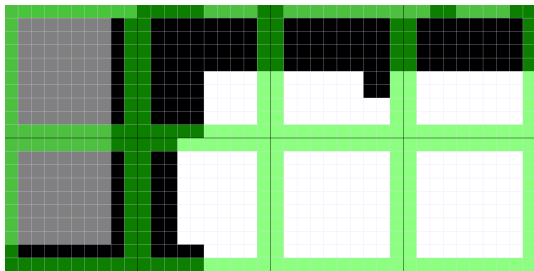


Abbildung 2.4: Roboterscanausschnitt.

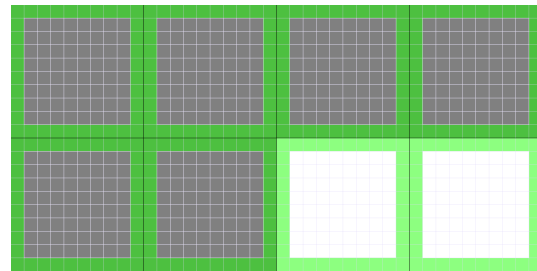


Abbildung 2.5: Resultierende Wabenkarte.

### 2.2.5 Der Regeltakt

Mit dem Verbindungsaufbau wird für jeden Roboter ein separater Thread gestartet. Nach der eingehenden Verbindung verweilt jeder Thread in einer Dauerschleife, die durch den Benutzer beendet werden kann und damit die Verbindung zwischen den Geräten trennt. Die Dauerschleife terminiert ebenfalls, wenn die Verbindung zum Roboter, durch einen Verbindungsabbruch oder Beenden des Clients, getrennt wird. Wird die Schleife beendet, so wird auch der Thread beendet und der Server ist bereit für eine erneute Verbindung mit einem Roboter.

Während des Schleifendurchlaufs werden im Regeltakt alle Operationen durchgeführt, um die erforderlichen Daten, die der Roboter benötigt, bereitzustellen und zu übertragen, sowie alle Informationen, die der Server benötigt, für die Darstellung, Speicherung und Weiterverarbeitung, bereitzustellen und zu setzen. Bevor die Schleife beginnt werden alle notwendigen Variablen, darunter fallen auch die RobotInformation und SearchInformation, initialisiert und anschließend der Beginn der Dauerschleife eingeleitet. Durch die Initialisierung wird festgelegt, dass der Client noch keinen neuen Pfad benötigt, deshalb wird im ersten Durchlauf der Schleife die Ermittlung eines Punktes durch die Koordination, die Generierung des Pfades der Bahnplanung und Bestimmung kollisionsfreier Fahrten der Kollisionsvermeidung übersprungen. Dies führt direkt zur ersten Nutzdatenübertragung zwischen Server und Client. Dabei erhält der Server die Positionsdaten des Roboters. Diese werden direkt in den entsprechenden Variablen gespeichert, um bei erneutem Zeichnen der Karte die aktuelle Position des Roboters darstellen zu können. An dieser Stelle endet die Schleife und der Prozess beginnt von vorne. Vor dem Neubeginn der Schleife wird vorweg noch eine Sleep eingefügt, der die Threadkontrolle abgibt und dem System erlaubt während dieser Wartezeit andere Operationen durchzuführen. Sollte der Client einen neuen Pfad angefordert haben, wird dieser, im nächsten Durchlauf, mittels der Koordination, Bahnplanung und Kollisionsvermeidung generiert (Kapitel 2.2.6). Der erstellte Pfad sowie die Länge des zu übergebenden Pfades und der Status der Suche werden in den entsprechenden Strukturen gespeichert und anschließend übermittelt. Schließlich ergibt sich aus dem schematischen Programmcode aus 2.7 der in Abbildung 2.6 dargestellte Verlauf für den Regeltakt.



```

1 #define SLEEPTIME 500
2 Point point;
3
4 while(holdConnection)           //Dauerschleife
5 {
6     if(needPathTable)
7     {
8         //Punkt durch Koordination ermitteln
9         point = GetNextPoint(pSelfRob->robotId);
10
11         //Pfad durch Bahnplanung ermitteln
12         CreateNextPath(robotId, robot->position ,point, &internalPathList);
13
14         //Kollisionen verhindern
15         avCollision->savePath(robot->position, &internalPathList);
16     }
17
18     //Nutzdaten"ubertragung
19     connection->Transmit(robotInformation, searchInformation, internalPathList);
20
21     //Abgabe der Threadkontrolle
22     Sleep(SLEEPTIME);
23
24     //Abbruch einleiden
25     holdConnection = changeConnectionStatus();
26 }

```

Listing 2.7: Thread des Regeltaktes.

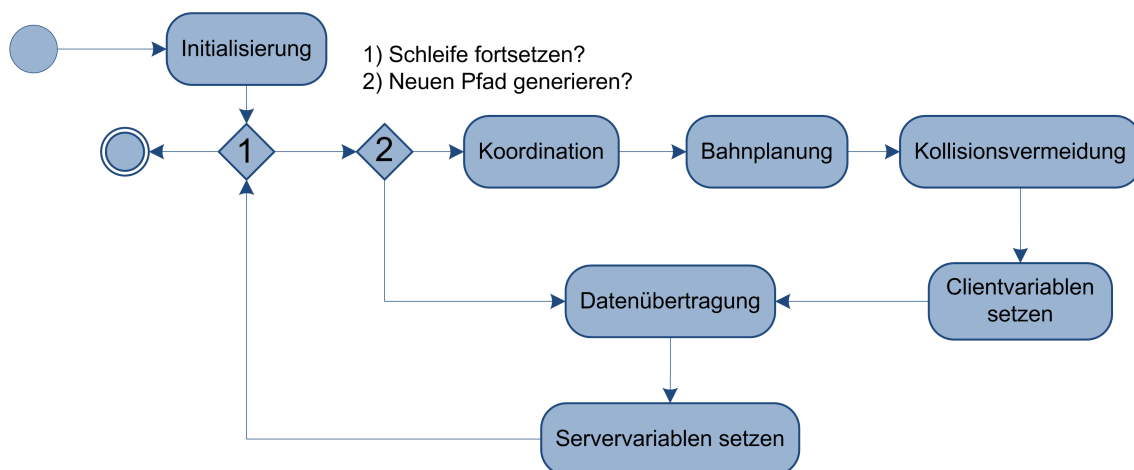


Abbildung 2.6: Programmablauf des Roboter-Threads.

Die Länge der Wartezeit nach einer Schleife kann vom Benutzer im Programm festgelegt werden. Während der Versuche hat sich die Wartezeit von 500ms als gutes Maß bewährt. Die Zeit  $t_{Takt}$  in der eine Übertragung pro Thread erfolgt ergibt sich demnach aus:

$$t_{Takt} = t_{Operationen} + t_{Warten} \quad (2.1)$$

mit  $t_{Operationen}$  als die Zeit die benötigt wird um die Operationen Koordination, Bahnplanung und Kollisionsvermeidung durchzuführen und  $t_{Warten}$  als feste Wartezeit, die vom Benutzer angegeben wird. Da die Zeit  $t_{Operationen}$  für jeden Takt unterschiedlich ist, ist folglich auch die Zeit des Regeltakts unregelmäßig.

### 2.2.6 Koordination, Bahnplanung und Kollisionsvermeidung

#### Koordination

Der Weg, den die Roboter während der Suche zurücklegen, wird durch die Koordination, Bahnplanung und Kollisionsvermeidung berechnet und festgelegt. Die Koordination ermittelt einen neuen Punkt im Raum, den der Roboter als nächstes anfahren soll. Dies geschieht global. Das bedeutet, dass alle Roboter gemeinsam nach dem gleichen Koordinationsalgorithmus fahren. Strategie für eine Koordination wäre beispielsweise eine Discovery-Map, in der die Punkte eingetragen sind, die die Roboter schon besucht bzw. erkundet haben und die Punkte, die noch nicht erkundet wurden müssen als nächstes angefahren werden. Zurzeit sind zwei sehr einfache Koordination-Strategien implementiert, da komplexe Strategien in dem zur Verfügung stehenden Suchterrain nicht nötig sind. In der ersten Variante werden fest eingespeicherte Punkte im Raum angefahren, die jedoch für jede Karte neu eingetragen werden müssen. Die andere Strategie basiert auf Zufallspunkten, die während der Laufzeit erstellt werden. Dabei wird eine zufällige Koordinate erstellt und überprüft, ob diese auch auf einem befahrbaren Punkt liegt. Sollte der generierte Punkt in einem Hindernis liegen, wird so lange ein neuer Punkt generiert, bis der Punkt nicht im Hindernis liegt.

#### Bahnplanung

Mit dem Punkt der Koordination wird das neue Ziel des Roboters generiert, das als nächstes erreicht werden muss. Dazu berechnet die Bahnplanung einen Weg durch den Raum, den der Roboter abfahren soll. Im Gegensatz zur Koordination, kann jeder Roboter eine unterschiedliche Bahnplanung besitzen. Dazu stehen verschiedene Strategien zur Auswahl. Beispielsweise ein RRT (Rapidly-exploring random tree), der zufällig Punkte im Raum generiert und diese zu einem Weg vom Roboter zum Zielpunkt verbindet. Ziel ist es durch die Bahnplanung eine Liste von Punkten zu erhalten, die der Roboter kollisionsfrei durch Rotation und Vorwärtsbewegung, erreichen muss. In der aktuellen Version wurde eine selbst entwickelte Methode implementiert, die „Fast Sweep 4“ genannt wurde und dem A-Stern Algorithmus ähnelt. Diese Methode startet am Punkt an dem sich der Roboter befindet und breitet sich von Schritt zu Schritt über die Karte aus, bis das Ziel erreicht wurde. Dabei werden in jedem Schritt die Entfernungen zum Roboter in den leeren anliegenden Zellen der Karte gespeichert, die beim Erreichen des Punktes, durch

Rückwärtslaufen, zu einem Pfad erstellt werden. Als Ergebnis dieser Funktion entsteht eine Liste von Punkten, die durch eine Interpolation die unnötigen Zwischenpunkte entfernt und daraus die benötigte Pfad-Liste resultiert.

### Kollisionsvermeidung

Neben der Kollision, die mit der Umgebung entstehen kann, muss bei einer Suche mit mehreren Robotern auch sichergestellt werden, dass keine Kollision untereinander entsteht. Dafür stand beispielsweise eine Strategie zur Verfügung, die einer Methode der Bahntechnik ähnelt. Dazu wird nach dem Erreichen eines Punktes überprüft, ob der nächste Fahrabschnitt sicher ist und eine Fahrstraße gestellt werden kann, die zu dem Zeitpunkt von keinem anderen Roboter genutzt werden kann. Die implementierte Methode sperrt nach der Bahnplanung den kompletten Raum, der von einem Roboter abgefahren werden soll, in dem die Punkt der Karte als Hindernis für die Bahnplanung der andren Roboter eingestellt werden. Erreicht der Roboter die Punkte der Bahnplanung, wird der bereits bewältigte Raum wieder für alle Roboter freigegeben.

#### 2.2.7 Erweiterung von Koordination und Bahnplanung

Die Struktur des Programms ist so konstruiert, dass durch ein hinzufügen weniger Programmcodezeilen neue Koordinationsalgorithmen und Bahnplanungsalgorithmen aufgenommen werden können. Koordinationen, die hinzugefügt werden sollen sind Klassen, die von der Klasse „Coordination“ abgeleitet werden müssen. Das Ableiten bewirkt den Zwang die virtuelle Methode *GetNextPoint(..)* zu implementieren, die im Programmverlauf vom Roboter aufgerufen wird und den neuen Zielpunkt für den Roboter ermittelt und zurückgibt. Zur Berechnung des Punktes steht der Koordination lediglich die Karte des Raumes zur Verfügung. Das Header-Template einer solchen Klasse ist in Listing 2.8 dargestellt.

```
1 #pragma once
2 #include "coordination.h"
3
4
5 class Coordination_None : public Coordination
6 {
7 public:
8     Coordination_None(void);
9     ~Coordination_None(void);
10
11     Point GetNextPoint(int robotId);
12 };
```

Listing 2.8: Koordinationstemplate.

Die Integration einer neuen Bahnplanung ist identisch zu der Integration einer Koordination. Bahnplanungen müssen von der Klasse „Pathplan“ abgeleitet werden, damit sichergestellt wird, dass die Funktion *CreateNextPath(..)* vorhanden ist. Diese berechnet durch die Angabe von Startpunkt und Zielpunkt eine neue List von Wegpunkten für den Roboter. Die Liste befindet sich nach der Funktion in der übergebenen Referenz „InternalPathInformation“. Listing 2.9 entspricht dem Header-Template der Klasse.

```

1 #pragma once
2 #include "Pathplan.h"
3
4 class Pathplan_None : public Pathplan
5 {
6 public:
7     Pathplan_None();
8     ~Pathplan_None();
9
10    int CreateNextPath(int robotId, Point start,
11                      Point end, InternalPathInformation *ipi);
12 };

```

Listing 2.9: Bahnplantemplate.

Der angezeigte Name der Koordination und Bahnplanung sehen in den Variablen „coordinationName“ und „pathplanName“ die in den jeweiligen Basisklassen enthalten sind. Sind die Klassen im Programm integriert, müssen diese noch im Programm angemeldet werden. Dies geschieht durch das Inkludieren der Header-Dateien in der „stdafx.h“ wie in Listing 2.10 und das Erstellen der Instanz in der Datei „PathManager.cpp“, wie es in Listing 2.11 gezeigt wird.

```

1 //UserInput:-----//User muss seine neu erstellte *.h Datei hier einf"ugen!
2 #include "Coordination.h"
3 #include "Coordination_None.h"
4 #include "Coordination_Random.h"
5 #include "Coordination_Fixpoint.h"
6
7 #include "Pathplan.h"
8 #include "Pathplan_None.h"
9 #include "Pathplan_FastSweep4.h"

```

Listing 2.10: Anmeldung der Klassen in stdafx.h.

```

1
2 //UserInput: User muss die Anzal der Coordinationen Manuell eintragen
3 numberOfCoordinations = 3;
4 coord = new Coordination*[numberOfCoordinations];
5 for(int i=0; i<numberOfCoordinations; i++)
6 {
7     coord[i] = NULL;
8 }
9
10 //UserInput: Koordinationsklassen hinzuf"ugen
11 coord[0] = new Coordination_None();
12 coord[1] = new Coordination_Fixpoint();
13 coord[2] = new Coordination_Random();
14

```

```
15 //UserInput: User muss die Anzahl der Bahnplaner Manuell eintragen
16 numberOfPathplaner = 2;
17 pathplan = new Pathplan*[numberOfPathplaner];
18 for(int i=0; i<numberOfPathplaner; i++)
19 {
20     pathplan[i] = NULL;
21 }
22 //UserInput: Bahnplanung hinzufügen
23 pathplan[0] = new Pathplan_None();
24 pathplan[1] = new Pathplan_FastSweep4();
```

Listing 2.11: Anmeldung der Klassen in PathManager.cpp.

## 2.3 Client

Der Client in unserer Client-Server-Architektur ist für das Anfahren der einzelnen Routenpunkte und die Erkennung des Balls zuständig. Dazu haben wir die uns zu Verfügung gestellten Implementierungen eines Sonar-Partikelfilters und der Ballerkennung verwendet. Wir beschreiben im Folgenden zunächst, wie wir den Partikelfilter und die Ballerkennung genutzt haben. Anschließend wird die Funktionsweise des Clients beschrieben.

### 2.3.1 Ballerkennung

Die Ballerkennung läuft in einen eigenen Thread im Client, da sie und die Fahrschleife des Roboters sich gegenseitig blockieren würden. Dabei greift sie auf das Kamerabild einer Logitech Webcam Pro 9000 zurück. Nach Initialisierung der Ballerkennung werden folgende Schritte durchgeführt, bis der Ball gefunden oder die Suche abgebrochen wurde:

Visual Paradigm for UML Community Edition [not for commercial]

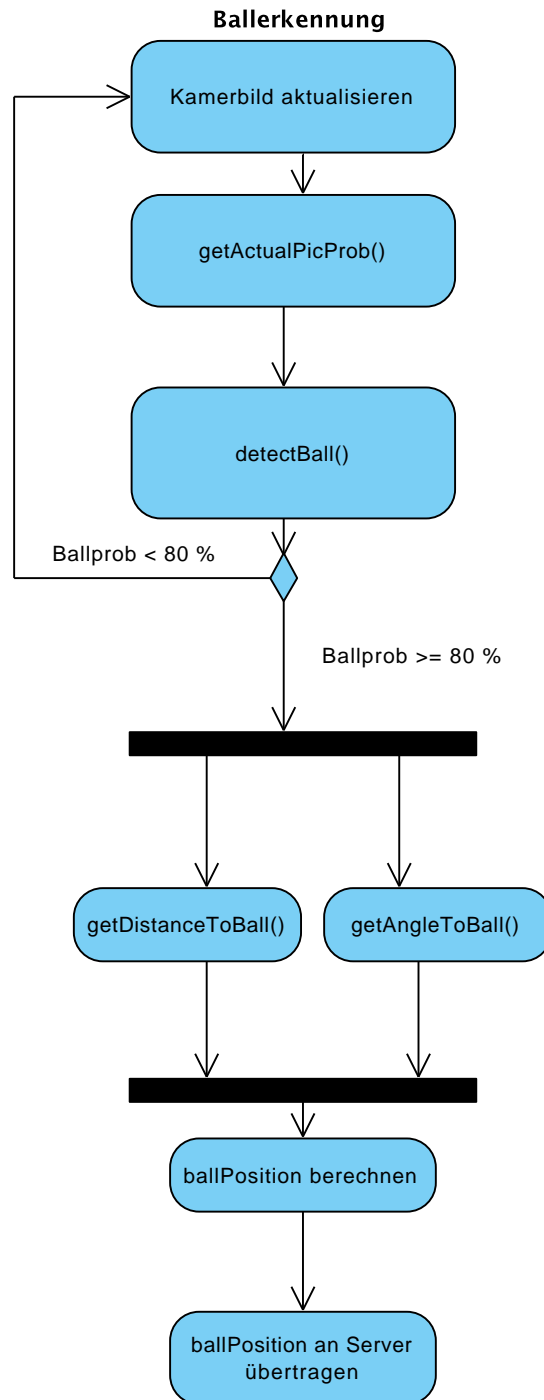


Abbildung 2.7: Ablauf der Ballerkennung

Die Berechnung der Ballposition arbeitet dabei wie folgt:

Mit den Methode `getDistanceToBall()` und `getAngleToBall()` wird die Entfernung (in mm) und der Winkel zum Ball (in Grad von der Mitte des Bildes aus ( $0^\circ$ ), links positiv, rechts negativ) abgerufen. Aus den

Ergebnissen wird zusammen mit der aktuellen Position des Roboters die Ballposition für die Anzeige auf der Karte des Servers bestimmt, dazu wird zuerst der Ausrichtungsvektor des Roboters um den Winkel zum Ball gedreht. Dann wird die Entfernung zum Ball ermittelt, da diese in Millimeter (mm) ist, muss die Entfernung in Pixel (px) umgerechnet werden, um die Position auf der Serverkarte korrekt zu visualisieren. Dazu dient der Wert `pixelPerMm` aus der Konfiguration, welcher die Auflösung der Karte in px/mm angibt. Um nun die Ballposition zu erhalten, wird der gedrehte Ausrichtungsvektor (ein Einheitsvektor) mit `pixelPerMm · Entfernung` zum Ball skalar multipliziert und zum aktuellen Positionsvektor addiert. Danach wird die gefundene Position zum Server übertragen.

Wie bereits erwähnt greifen wir bei der Ballerkennung auf eine Bachelorarbeit zurück. Dadurch konnten wir einfach die API der dabei entwickelten Software nutzen, ohne uns selber mit den Details der Ballerkennung beschäftigen zu müssen. Sie verfolgt folgenden Ansatz: Im Kamerabild wird nach roten Pigmenten gesucht. Erreichen diese eine gewisse Anzahl, sucht die Software nach kreisförmigen Objekten, die diese Pigmente enthalten. Je nachdem, wie weit diese Kriterien erfüllt sind, berechnet die Software eine Wahrscheinlichkeit, ob sich im aktuellen Bild ein Ball befindet und die Entfernung und den Winkel zum vermuteten Ball. Um vernünftige Werte zu erhalten ist es unabdingbar, die Software vorher zu kalibrieren<sup>1</sup>. Nichts destotrotz kann es zu False-Positives kommen. Näheres zu den Ursachen von False Positives finden sich im Abschnitt 5 ab Seite 37.

### 2.3.2 Sonar-Partikelfilter

Er nutzt das im Roboter integrierte Sonarsystem, um anhand einer Karte die Position des Roboters im Raum zu bestimmen. Dazu bedienen wir uns seiner Klasse `particleSet`. Mit dem Konstruktor

```
1 particleSet::particleSet(
2     const char *map_path, const char *wskTxT_path,
3     bool use_ray_casting, int max_x, int min_x,
4     int max_y, int min_y, int map_size, int map_res,
5     int numofParticle, ArRobot *p3dx
6 )
```

erzeugen wir ein Objekt `particleSet` mit der Partikelmenge. Dabei übergeben wir ihm auch den Pfad zur Karte des Raumes und legen fest, ob wir Raycasting benutzen oder nicht, sowie die Auflösung und Koordinaten der Karte.

Am Partikelfilter haben wir ein paar Änderungen vorgenommen. Es gab keine Möglichkeit, von außerhalb des Filters auf die Koordinaten und die Ausrichtung des Roboters zuzugreifen. Dazu wurden die Methoden `StartFilter()` und `findBestParticle()` so erweitert, dass die aktuelle Position des Roboters

<sup>1</sup>Die genaue Vorgehensweise wird im Abschnitt 4.1 Nutzung ab Seite 29 beschrieben

und seine Ausrichtung als Zeiger auf das erste Element eines double-Arrays ([x,y,Theta, Partikelwsk.]) zurückgegeben wird. Da dieses in `findBestParticle()` dynamisch mit `new` alloziiert wird, muss es, wenn es nicht mehr benötigt wird, mit `delete[]` gelöscht werden. Außerdem ist nun möglich, die Anzahl der Sonare von außerhalb des Filters ohne neukompillieren einzustellen. Die Visualisierungen wurden aus Performancegründen entweder entfernt oder deaktiviert. Auch wurde ein Speicherleck behoben, welches dadurch zustande kam, dass Measure-Objekte nur alloziiert wurden und, nachdem sie nicht mehr benötigt wurden, nicht dealloziiert wurden. Zudem wurde, für eine einfachere Auswertung und weil die Partikelvisualisierung öfter zu Abstürzen unseres Programms geführt hat, der Filter um eine Funktion erweitert, die, sofern dies beim Kompilieren der Software aktiviert wurde, bei jedem Filtern alle Partikel als Tripel (x,y,Partikelwsk.) in eine Datei schreibt. Die Datei hat dabei als Prefix eine Zahl im Namen, die bei 0 beginnend nach jedem Filtern um 1 erhöht wird. Passend dazu wurde ein Visualisierer geschrieben, der die Partikelmenge grafisch darstellt, auf den noch später eingegangen wird.

### 2.3.3 Funktionsweise des Clients

Der Client selber arbeitet folgendermaßen:

Beim Start wird die Konfiguration aus der Datei `config.cfg` eingelesen, wenn diese nicht gefunden wird, nutzt der Client die fest eingestellten Standardwerte. Die `config.cfg` hat folgende Struktur:

```
1 #Kommentarzeile (wird ignoriert)
2 Sonaranzahl als int
3 #Kommentarzeile (wird ignoriert)
4 COM-Port, an dem der Roboter angeschlossen ist, als String
5 #Kommentarzeile (wird ignoriert)
6 Schrittweite fuer die Anlerndrehungen als int in Grad
7 #Kommentarzeile (wird ignoriert)
8 Anzahl der 360Grad Drehungen als int
9 #Kommentarzeile (wird ignoriert)
10 Server-IP als String
11 #Kommentarzeile (wird ignoriert)
12 Pixel in der Karte pro mm als double
```

Listing 2.12: Aufbau der `config.cfg`

Ein Beispielkonfiguration für den Fahrstuhlvorraum vor dem Robotik-Labor im Braunschweiger Informatikzentrum wäre z. B.

```
1 #Anz Sensoren
2 8
3 #COM Port
4 COM5
5 #Schrittweite fuer die Anlerndrehungen
6 15
7 #Anzahl der 360Grad Drehungen
8 1
9 #Server-IP
10 134.169.36.241
11 #Pixel pro mm
```



12 0.05081

## Listing 2.13: Ein Beispiel für eine komplette config.cfg

Die Standardwerte sind (16, COM3, 45, 0, 127.0.0.1, 1).

Danach wird die Verbindung zum Roboter hergestellt und der Partikelfilter mit 10000 Partikeln initialisiert und das erste Mal gefiltert. Danach erfolgt die konfigurierte Anzahl an Drehungen in der konfigurierten Schrittweite um die erste Positionierung zu verbessern. Nach jedem Drehschritt wird einmal gefiltert. Nach den Drehungen wird noch einmal gefiltert und der Rückgabewert als Startposition genommen. Erst jetzt wird die Verbindung zum Server hergestellt und die Roboterposition übermittelt, sowie ein Thread gestartet, der regelmäßig die Structs `RobotInformation`, `PathInformation` und `SearchInformation` mit dem Server abgleicht. Sie enthalten die zum Steuern des Clients, sowie zur Durchführung der Suche notwendigen Informationen. Nach Verbindungsaufbau wartet der Client bis der Server die Suche startet. Sobald die Suche startet, wird ein Thread zur Ballerkennung gestartet und der Client geht in seine Fahrschleife über, die erst endet, wenn der Ball gefunden wurde oder stoppt, wenn der Server die Suche pausiert oder beendet. Solange die Fahrschleife aktiv ist, wartet der Client auf den nächsten anzufahrenden Punkt vom Server, sobald er diesen erhalten hat fährt der Roboter den Punkt an.

Visual Paradigm for UML Community Edition [not for commercial use]

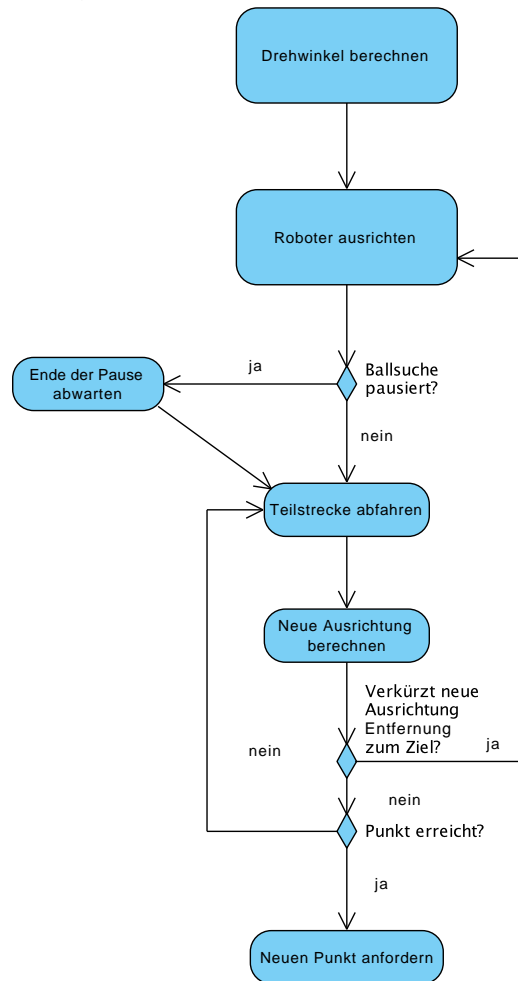


Abbildung 2.8: Ablauf der Fahrschleife des Clients

Dies geschieht schrittweise:

Zuerst wird die Richtung und der Drehwinkel der Drehung für eine Ausrichtung zum Punkt berechnet. Dazu wird der Winkel zwischen dem Ausrichtungsvektor des Roboters und dem Vektor vom Roboter zum Punkt berechnet und dann geprüft, ob der Vektor zwischen Roboterposition und dem Punkt um den ermittelten Winkel im Uhrzeigersinn oder gegen den Uhrzeigersinn von der RoboterAusrichtung aus liegt um die Drehrichtung zu ermitteln. Dann wird der Roboter entsprechend der ermittelten Werte zum Punkt hin gedreht. Dann wird, falls die Suche gerade pausiert, gewartet bis der Server die Pause beendet. Danach wird  $\text{drivingTime} * \text{driveLengthWeight}$  ms gefahren, wobei  $\text{drivingTime}$  die Fahrzeit pro Schritt ist und  $\text{driveLengthWeight}$  ein Anpassungsfaktor der Fahrzeit ist für den Fall, dass die Reststrecke in weniger als der Fahrzeit pro Schritt zurückgelegt werden kann. Dazu wird aus den gefahrenen Strecken in jedem Fahrschritt und der jeweiligen Fahrzeit die Geschwindigkeit in px pro ms berechnet und über die Schritte der exponentiell geglättete Mittelwert berechnet, woraus dann  $\text{driveLengthWeight}$  für die Rest-

strecke berechnet wird, wenn die zu klein für die normale Fahrzeit ist. Dann wird eine neue Ausrichtung zum Punkt berechnet und mit der aktuellen verglichen, verkürzt sich dadurch die Entfernung zum Ziel, wird der Roboter neu ausgerichtet, ansonsten wird die alte Ausrichtung beibehalten. Danach wird wieder eine Teilstrecke gefahren usw. bis sich dem Punkt auf einen Abstand unterhalb einer Toleranzschwelle angenähert wurde (`delta_pos`). Die Toleranzschwelle ist nötig, da aufgrund der Ungenauigkeiten des Sonar-Partikelfilters nicht immer ein genaues Erreichen der Zielposition gewährleistet werden kann. Auf die prinzipiellen Ungenauigkeiten des Partikelfilters wird näher im Abschnitt 5 Evaluation ab Seite 37 eingegangen. Danach wird gewartet, bis der Server das Erreichen des Punktes registriert hat und ein neuer, anzufahrender Punkt übermittelt wurde.

## 3 Erstellung der Software

In diesen Abschnitt beschreiben wir den Erstellungsprozess der Software. Dazu beschreiben wir erst die Struktur des Projektes, den Aufbau des Buildsystems für den Client und schließlich den Erstellungsprozess für Client und Server.

### 3.1 Aufbau des Projektbaums

Der aus dem SVN ausgecheckte Projektbaum ist wie folgt aufgebaut:

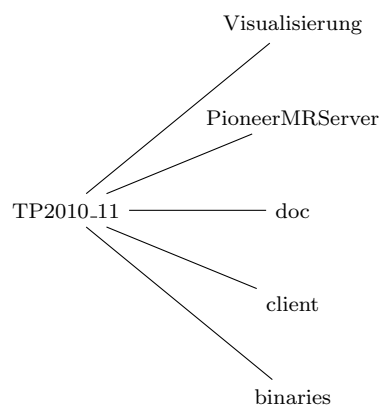


Abbildung 3.1: Projektbaum nach svn checkout

Da es beim Client nötig war, diverse Libraries des Institutes für Robotik und Prozessinformatik (IRP) der technischen Universität Braunschweig sowie den Sonar-Partikelfilter und die Ballerkennung zu integrieren, wurde dazu eine besondere Buildumgebung auf Basis von CMake geschaffen. Sie wird im Abschnitt 3.2 ab Seite 23 beschrieben.

Einen ersten Überblick über die Struktur des Ordners „client“ gibt folgende Baumübersicht:

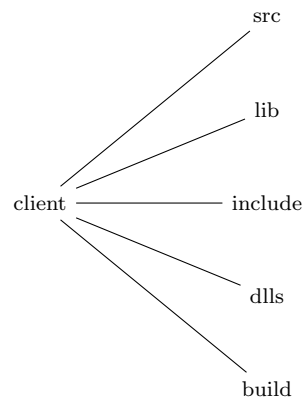


Abbildung 3.2: Projektbaum im Unterordner client

Der Ordner „build“ enthält die durch CMake generierten VisualStudio Projekte und fertig gebauten ausführbaren Dateien und DLLs. Die Ordner „dlls“ und „lib“ enthalten zur Erzeugung und Ausführung der Projekte nötige externe \*.dll und \*.lib Dateien. In den Ordnern „include“ und „src“ sind schließlich die Quelltexte und Header der verwendeten Libraries sowie unseres Clients „p3dxSteuerung“ enthalten.

Die Notwendigkeit einer gesonderten Buildinfrastruktur war beim Server (PioneerMRServer) nicht gegeben, da dieser nicht von den Institutsbibliotheken abhängt. Entsprechend reichte es dort, ein normales VisualStudio Projekt zu erstellen.

## 3.2 Integration bestehender Projekte in ein neues Buildsystem für den Client

### 3.2.1 Warum ein neues Buildsystem?

Die Ballerkennung und der Sonarpartikelfilter hängen von vielen Altprojekten ab. Das Beispiel für die Steuerung der Pioneer 3-DX hing zudem von der ARIA Bibliothek<sup>1</sup> ab. Bei den Altprojekten fanden sich auch zum Teil fest kodierte Pfade. Da auch die Konvertierung der Solutions in das VS 2010 Format meistens fehlschlug und auch ARIA nicht mit der mitgelieferten Solution unter VS 2010 gebaut werden konnte, haben wir uns für den Client für den Einsatz eines neuen Buildsystems entschlossen. Aufgrund der Einfachheit der Beschreibung der Buildvorgänge für die einzelnen zu integrierenden Projekte und dem Hinzufügen neuer Unterprojekte, sowie der Möglichkeit, nicht von einem konkreten Buildsystem abhängig zu sein sowie existierender Erfahrungen mit dem System, fiel die Wahl auf CMake<sup>2</sup>. Bei CMake handelt es sich um ein open-source Metabuildsystem. Es erzeugt Build-Vorschriften für andere Buildsysteme (u.a. GNU make, nmake, msbuild). Somit wäre es bei Bedarf auch Möglich gewesen, auf eine andere Visual Studio Version umzusteigen ohne große Änderungen vornehmen zu müssen.

<sup>1</sup><http://robots.mobilerobots.com/wiki/ARIA>

<sup>2</sup><http://www.cmake.org/>

### 3.2.2 Integration der Projekte

Zuerst wurde eine passende Ordnerstruktur angelegt: include für Header-Dateien, src für die cpp-Dateien der einzelnen Projekt, bin für eventuell zum Linken benötigte Kompillate, deren Quellen nicht in den Build-Prozess integrierbar waren und build als Zielordner für erzeugte Buildfiles.

Als nächstes wurde für das Projekt im Ordner des Clients eine CMakeLists.txt-Datei angelegt:

```

1 cmake_minimum_required (VERSION 2.6)
2 project (TP_WS_2010)
3
4 include_directories (include)
5 include_directories (include/ControlExample)
6 include_directories (include/erob/)
7 include_directories (include/Balldetection)
8 include_directories (include/p3dxSteuerung)
9 include_directories (include/StochasticLib)
10 include_directories (include/PioneerMRClient)
11 include_directories (include/Aria)
12 include_directories (include/OccupancyGridMap)
13 include_directories (include/irpVideo/DirectShow)
14 link_directories (${TP_WS_2010_BINARY_DIR})
15 link_directories (${TP_WS_2010_SOURCE_DIR}/lib/)
16 add_subdirectory (src)

```

Listing 3.1: CMakeLists.txt für das Buildsistem des Clients

Diese definiert den Namen des Projektes (wichtig für Referenzen auf z.b. das Stammverzeichnis des Projektes oder das Buildverzeichnis, da die entsprechenden Variablennamen mit dem Projektnamen + \_ als Präfix versehen werden. Zudem wurden die include-Verzeichnisse und zum Linken relevante Verzeichnisse festgelegt. Danach wurde das src-Verzeichnis als Projektunterordner hinzugefügt. Dadurch werden die Anweisungen in der CMakeLists.txt des Unterordners auch ausgeführt. Dies ermöglicht es, für die einzelnen Unterprojekte jeweils eine eigene kleine CMakeLists.txt zu verwenden.

Die CMakeLists.txt bindet nun alle Sourceordner der Unterprojekte ein:

```

1 %\begin{lstlisting}
2 cmake_minimum_required (VERSION 2.6)
3
4 add_subdirectory (irpUtils)
5 add_subdirectory (irpMath)
6 add_subdirectory (irpImage)
7 add_subdirectory (irpVideo)
8 add_subdirectory (irpFeatureExtraction)
9 add_subdirectory (alglib)
10 add_subdirectory (DistanceMap)
11 add_subdirectory (irpCamera)
12 add_subdirectory (irpV3d)
13 add_subdirectory (StochasticLib)
14 add_subdirectory (Visualization)
15 add_subdirectory (CamCalib)
16 add_subdirectory (Balldetection)
17 add_subdirectory (mathtest)
18 add_subdirectory (SonarParticleFilter)
19 add_subdirectory (PioneerMRClient)

```

```

20 add_subdirectory (OccupancyGridMap)
21 add_subdirectory (Balldetection_Test)
22 add_subdirectory (Aria)
23 add_subdirectory (p3dxSteuerung)
24 add_subdirectory (p3dxSteuerung_threaded)
25 add_subdirectory (calibrate)
26 #add_subdirectory (ControlExample)
27 add_subdirectory (utils)
28 add_subdirectory (kinect)
29 add_subdirectory (networkTest)

```

Listing 3.2: CMakeLists.txt für den Sourceordner

Bei den Unterprojekten gibt es zwei Arten: Bibliotheken und Anwendungen.

Die CMakeList.txt einer Bibliothek sieht wie folgt aus:

```

1 %\begin{lstlisting}
2 cmake_minimum_required (VERSION 2.6)
3
4 file(GLOB src "*.cpp")
5 file(GLOB includes "../include/irpCamera/*.h")
6
7 add_library (irpCamera ${src} ${includes})

```

Listing 3.3: CMakeLists.txt einer Bibliothek am Beispiel irpCamera

Die CMakeList.txt einer Anwendung sieht so aus:

```

1 cmake_minimum_required (VERSION 2.6)
2
3 file(GLOB src "*.cpp")
4 file(GLOB includes "../include/p3dxSteuerung/*.h")
5 add_executable (p3dxSteuerung_threaded ${src} ${includes})
6 target_link_libraries (p3dxSteuerung_threaded irpUtils MINPACK fftw rfftw
7 DirectShow irpFeatureExtraction irpMath alglib irpImage irpCamera irpVideo
8 irpV3d glew CamCalib Visualization StochasticLib OccupancyGridMap
9 SonarParticleFilter ws2_32 winmm advapi32 Aria DistanceMap Balldetection)

```

Listing 3.4: CMakeLists.txt des Clients

Zuerst werden alle cpp-Dateien zur Variablen `src` hinzugefügt, danach die Includes zur Variablen `include`. Je nachdem, ob es sich um eine Bibliothek oder Anwendung handelt, wird diese mit `add_library(name dateien)` oder `add_executable(name dateien)` hinzugefügt. Die Include-Dateien müssen dabei aber eigentlich nicht explizit hinzugefügt werden, dies geschieht nur, damit sie auch in einer VS Solution in der Dateiliste auftauchen. Bei Anwendungen können nun mit `target_link_libraries(exename libraries)` noch die zu linkenden Bibliotheken angegeben werden.

Es gab ein paar Probleme bei der Integration: zum einen mussten zuerst die Abhängigkeiten zwischen den Unterprojekten ermittelt werden, dies geschah mit Hilfe der originalen VS Solutions sowie durch Ausprobieren. Zum anderen gab es manchmal Dateien, die zwar in den Quellen lagen, jedoch in nicht in den Solutions auftauchten. Diese konnten den Buildprozess stören und mussten daher entfernt werden. Außerdem waren an ein paar Stellen kleinere Änderungen am Quellcode nötig, um ihn mit MSVC 2010

oder der Struktur unseres include-Ordners kompatibel zu machen.

### 3.2.3 Hinzufügen eines neuen Unterprojektes

Das Hinzufügen eines neuen Unterprojektes ist relativ einfach. Zuerst wird ein Unterverzeichnis für das Projekt im include-Ordner und eines im src-Ordner angelegt. Dann kopiert man die includes in den neuen Unterordner im include-Ordner. Dabei ist darauf zu achten, dass die Pfade in `#include`-Direktiven relative zum include-Ordner sein sollten. Danach kopiert man die cpp-Dateien in den neuen Unterordner im src-Verzeichnis.

Danach erstellt man in diesem Ordner eine `CMakeLists.txt` ähnlich wie oben, je nachdem ob es sich um eine Anwendung oder Bibliothek handelt. Man fügt zuerst die cpp-Dateien und Includes hinzu, fügt dann mit `add_library` oder `add_executable` Buildtargets hinzu und gibt eventuell noch zu linkende Bibliotheken bei Anwendungen an mit `target_link_libraries`.

## 3.3 Erstellung von Client und Server mit Visual Studio 2010

Da der Client mit Hilfe des im vorherigen Abschnitts auf CMake basierenden Buildsystems erzeugt wird, sind hier mehr Schritte als beim Server notwendig:

1. CMake für Windows (Version min. 2.6) herunterladen und installieren <sup>3</sup>
2. CMake GUI starten
3. Unter "Where is the source code" den Pfad des Client-Verzeichnisses eintragen und unter "Where to build the binaries" den Pfad zu dem Verzeichnis, wo die Solution für den Client erstellt werden soll
4. Auf "Configure" klicken, „Visual Studio 2010“ und „Use default native compilers“ auswählen, dann auf Finish klicken. Auf das Ende des Einrichtens warten.
5. Auf "Generate" klicken.
6. Im ausgewählten Zielverzeichnis befindet sich nun eine Visual Studio 2010 Solution (`TP_WS_2010.sln`).
7. Außerdem befindet sich im Unterordner „PioneerMRServer“ im Projektbaum eine Solution für den Server (`PioneerMRServer.sln`). Beide Solutions können ganz normal mit Visual Studio 2010 geöffnet und gebaut werden (F7 oder STRG-Alt-F7).

---

<sup>3</sup><http://www.cmake.org/>



## 4 Nutzung der Software

### 4.1 Benutzung der grafischen Benutzeroberfläche (GUI) des Servers

Zur Visualisierung der im Projekt implementierten Funktionen wurde mittels der objektorientierten Klassenbibliothek der Microsoft Foundation Class (MFC) eine Benutzeroberfläche erstellt. Abgeleitet von der Klasse "CFormView" dient das äußere Fenster als Gerüst für die benötigten Aktions- und Einstellungselemente, sowie für die Darstellung der Karte und des Protokollierungsfensters.

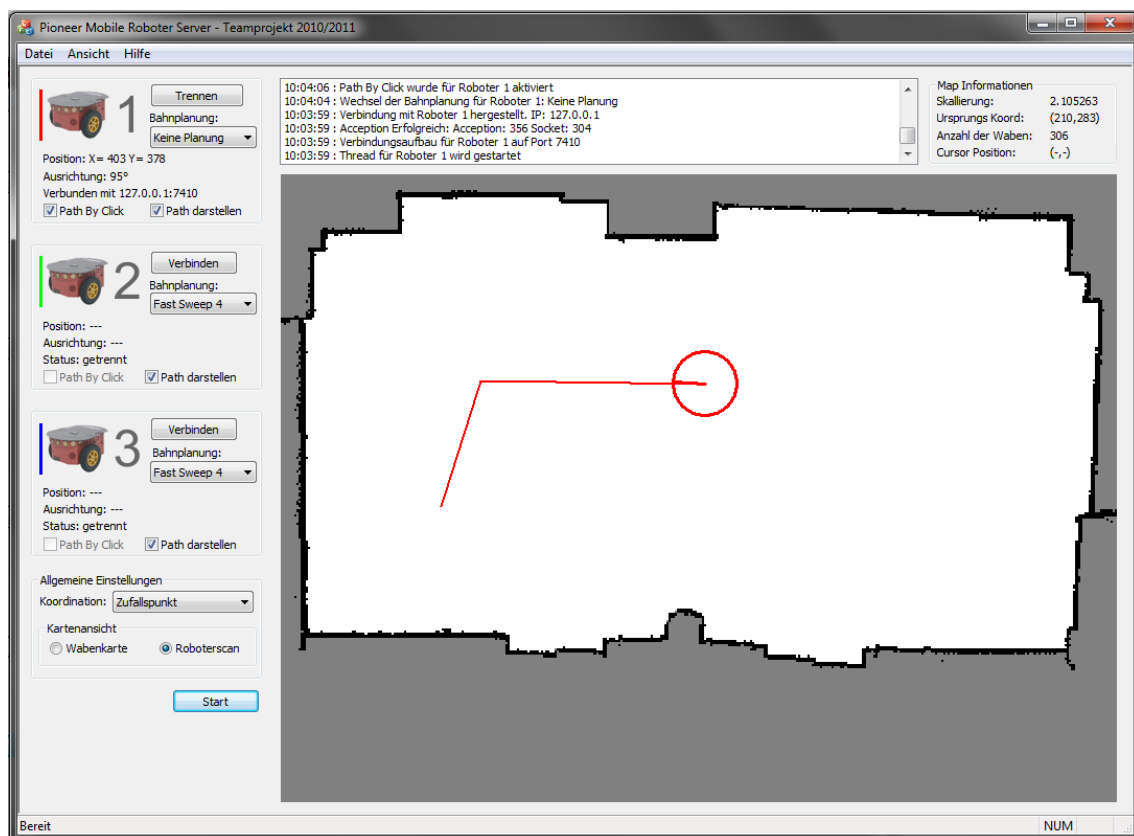


Abbildung 4.1: Pioneer Mobile Robot Server - GUI.

In der Abbildung 4.1 sind die Elemente des Programms zu sehen, die dem Benutzer die Steuerung ermöglichen und eine Übersicht des aktuellen Status geben. Auf der linken Seite der grafischen Oberfläche ist zu sehen, dass der Server bis zu drei mobile Roboter verwalten kann. Dargestellt werden die Roboter in ihren jeweiligen Farbe und mit ihrem abzufahrenden Pfad, falls diese Einstellung ausgewählt

wurde, auf der Karte, die den größten visuellen Teil der Benutzeroberfläche ausmacht. In der folgenden Abbildung 4.2 sind die spezifischen Robotereinstellungen und Roboterinformationen vergrößert dargestellt.



Abbildung 4.2: Robotereinstellungen und -informationen.

Durch das betätigen des „Verbinden/Trennen“-Buttons aus Abbildung 4.2 wird eine Netzwerkverbindung geöffnet und auf eine eingehende Verbindung eines Clients gewartet. Es kann nur jeweils auf eine Verbindung gewartet werden. In dieser Zeit werden die restlichen „Verbinden“-Buttons für den Benutzer deaktiviert. Sichtbar wird ein eingehende Verbindung, wenn der Button von „Abbrechen“ auf „Trennen“ wechselt und die deaktivierten Buttons wieder aktiviert sind. Verbindet sich ein Roboter mit dem Server werden die Positionsinformationen übertragen und mit der Position und Ausrichtung im entsprechenden Feld dargestellt. Als weiterer Hinweis wird zudem noch die entsprechende IP-Adresse und der Port der eingehenden Verbindung angegeben, um den verbundenen Rechner zu identifizieren und damit die dargestellten Roboter auf der Oberfläche den realen Robotern zuzuordnen. Für jeden Roboter kann im Dropdown-Menu mit der Beschriftung „Bahnplanung“ eine separate Bahnplanung ausgewählt werden, die vom Server zu gegebenen Zeitpunkten berechnet wird (siehe Kapitel Bahnplanung 2.2.6). Mit der Aktivierung der Option „Path by Click“ kann der Benutzer manuell, mittels Klicken in die Karte, einen Pfad für den Roboter bestimmen. Zur Darstellung des Pfads, der durch die Bahnplanung oder mittels „Path by Click“ erstellt wurde, muss die ComboBox „Path darstellen“ aktiviert sein.

In Abbildung 4.3 sind die allgemeinen Einstellungen vergrößert dargestellt. Die allgemeinen Einstellungen beinhalten die Auswahl der Koordination (siehe Kapitel 2.2.6), die im DropDown-Feld ausgewählt werden kann, sowie die Möglichkeit zur Darstellung der Karte, durch Selektion der Radio-Buttons, zwischen der Wabenkarte und dem Roboterscan (siehe Kapitel 2.2.4). Der „Start/Stop“-Button startet und beendet die Suche. Diese Einstellung verhindert ein sofortiges Starten des Suchvorgangs nach einer eingehenden Verbindung.

In Abbildung 4.4 sind die globalen Informationselemente aufgeführt, wie sie in Abbildung 4.1 im obe-

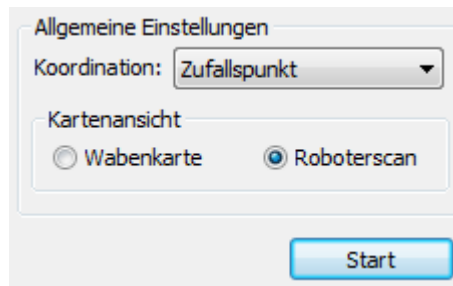


Abbildung 4.3: Allgemeine Einstellungen.



Abbildung 4.4: Globale Informationselemente.

ren Fensterbereich zu sehen sind. Der Bereich, der die Debuginformationen und Benutzernachrichten darstellt, ist eine modifizierte ListBox, die ursprünglichweise als Auswahlfeld verwendet wird. Jedoch wurde diese Option für die verwendeten Zwecke deaktiviert. Die Informationen der dargestellten Karte werden innerhalb des Gruppenrahmen „Map Information“ angezeigt. Darunter fällt die Skalierung der angezeigten Karte, die Koordinate des Ursprungs der angezeigten Karte, die Anzahl der in der Karte befindlichen Waben und die Cursor Position auf die der Mauszeiger besitzt, wenn dieser auf einen Punkt der Karte zeigt. Die Informationen der Karte sind in Kapitel 2.2.4 auf Seite 9 nachzulesen.

## 4.2 Client

### 4.2.1 Kalibrierung der Balldetection

Die Ballerkennung muss zunächst kalibriert werden, da ansonsten die ermittelte Entfernung zum Ball nicht korrekt ist. Dies ist deshalb nötig, da die Ausrichtung der auf dem Roboter montierten Webcam entscheidenden Einfluss auf die von der Ballerkennung ermittelten Werte hat. Zur VisualStudio Solution „TP2010/11“ gehört auch ein Unterprojekt calib. Nach Erstellung der ausführbaren Datei calib.exe ist diese an folgende Stelle des Projektbaums zu kopieren: `\client\src\calibrate\`. Für die folgenden Schritte werden dann ein Schachbrett sowie ein Maßband oder Ähnliches benötigt. Danach geht man folgendermaßen vor:

- Man stellt das Schachbrett und den Roboter in einen Abstand von ca 1 Meter voneinander entfernt so auf, dass die Kamera auf die Mitte des Schachbrettes zeigt.
- Danach misst man mit dem Maßband die Entfernung zwischen Roboter und Schachbrett, sowie den

Abstand vom Schachbrett zum Boden (im Normalfall 25 cm). Die beiden Enden der Entfernungsstrecke zeigt folgendes Bild:

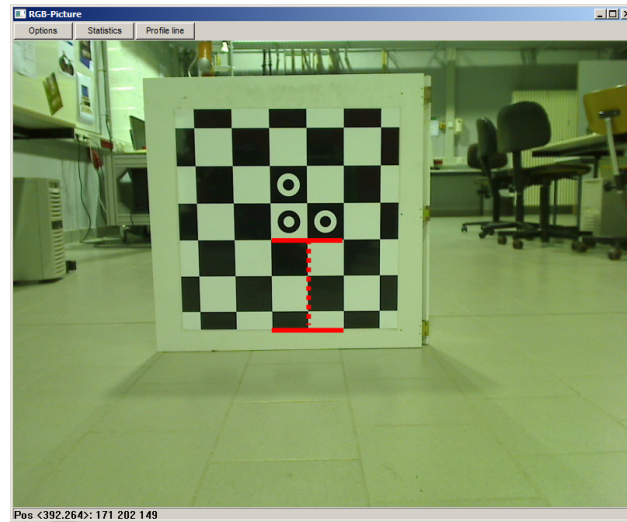
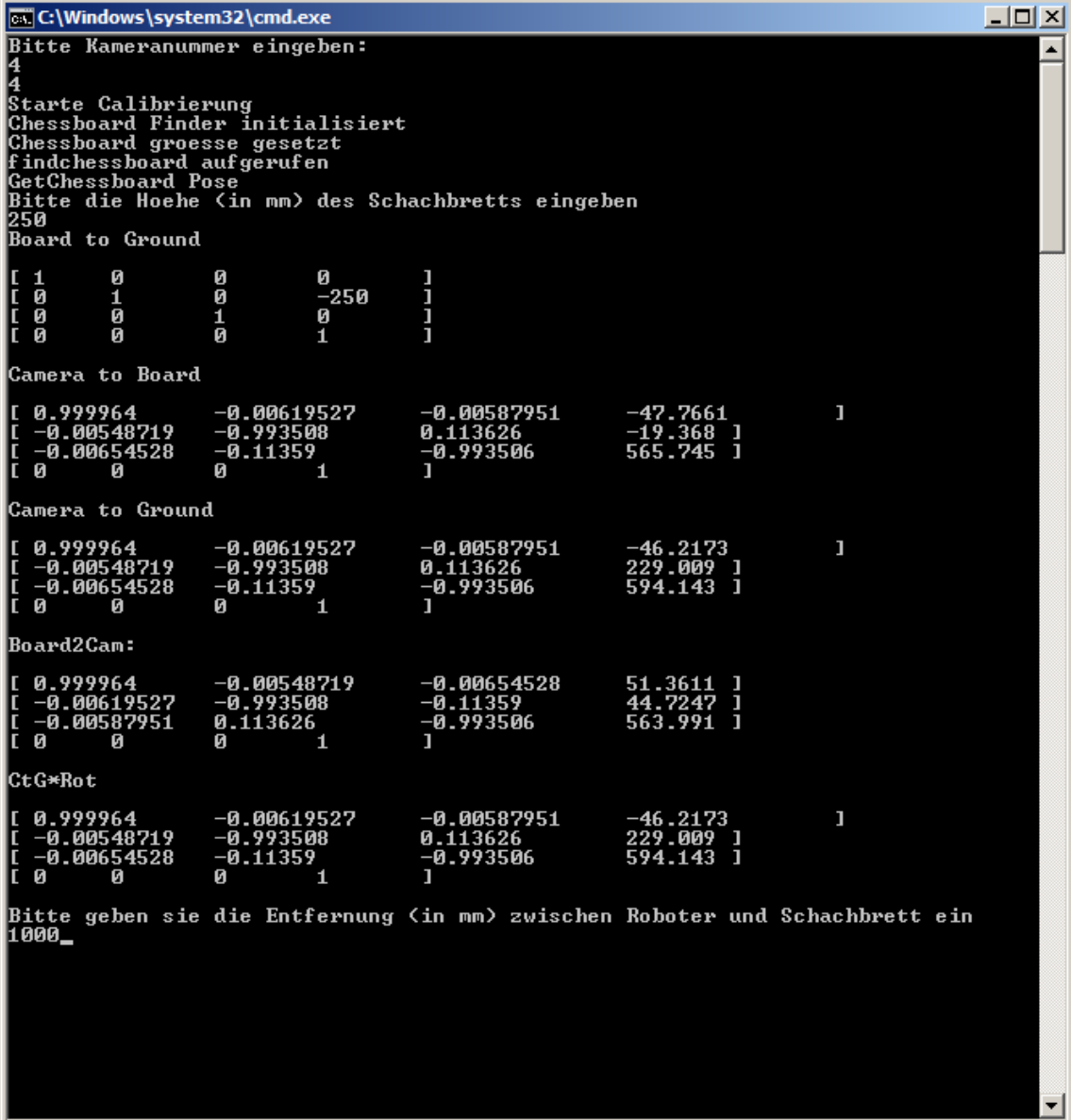


Abbildung 4.5: Abstand zwischen Boden und Schachbrett

- Anschließend startet man die Batchdatei `calibrate.cmd` im Ordner `\client\src\calibrate\`. Die Batchdatei fragt nun die Kameranummer, sowie die im vorherigen Schritt ermittelten Entfernungen in mm ab und startet anschließend durch Drücken der Enter-Taste die Erkennung:



```

C:\Windows\system32\cmd.exe
Bitte Kameranummer eingeben:
4
4
Starte Kalibrierung
Chessboard Finder initialisiert
Chessboard groesse gesetzt
findchessboard aufgerufen
GetChessboard Pose
Bitte die Hoehe <in mm> des Schachbretts eingeben
250
Board to Ground
[ 1 0 0 0 ]
[ 0 1 0 -250 ]
[ 0 0 1 0 ]
[ 0 0 0 1 ]

Camera to Board
[ 0.999964 -0.00619527 -0.00587951 -47.7661 ]
[ -0.00548719 -0.993508 0.113626 -19.368 ]
[ -0.00654528 -0.11359 -0.993506 565.745 ]
[ 0 0 0 1 ]

Camera to Ground
[ 0.999964 -0.00619527 -0.00587951 -46.2173 ]
[ -0.00548719 -0.993508 0.113626 229.009 ]
[ -0.00654528 -0.11359 -0.993506 594.143 ]
[ 0 0 0 1 ]

Board2Cam:
[ 0.999964 -0.00548719 -0.00654528 51.3611 ]
[ -0.00619527 -0.993508 -0.11359 44.7247 ]
[ -0.00587951 0.113626 -0.993506 563.991 ]
[ 0 0 0 1 ]

CtG*Rot
[ 0.999964 -0.00619527 -0.00587951 -46.2173 ]
[ -0.00548719 -0.993508 0.113626 229.009 ]
[ -0.00654528 -0.11359 -0.993506 594.143 ]
[ 0 0 0 1 ]

Bitte geben sie die Entfernung <in mm> zwischen Roboter und Schachbrett ein
1000_

```

Abbildung 4.6: Abfrage der zur Kalibrierung nötigen Parameter

- Die Kalibrierung ist erfolgreich, wenn folgendes Muster im Kamerabild angezeigt wird:

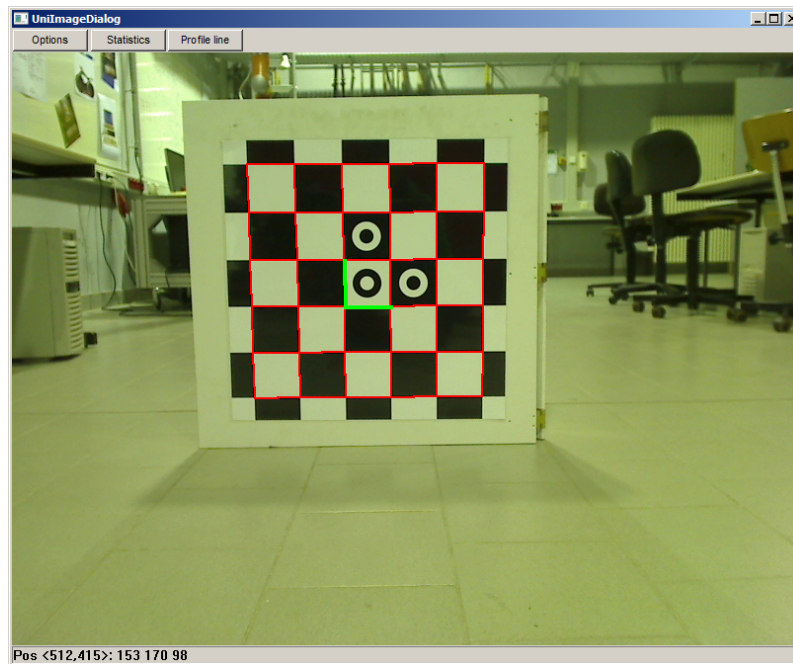


Abbildung 4.7: Darstellung einer erfolgreichen Kalibrierung der Ballerkennung

- Dann ist im Batch-Skript die Ausführung des Kalibriertools mit <STRG >-C zu unterbrechen. Auf die Nachfrage, ob auch die Batchdatei unterbrochen werden soll, gibt man „n“ ein:

```

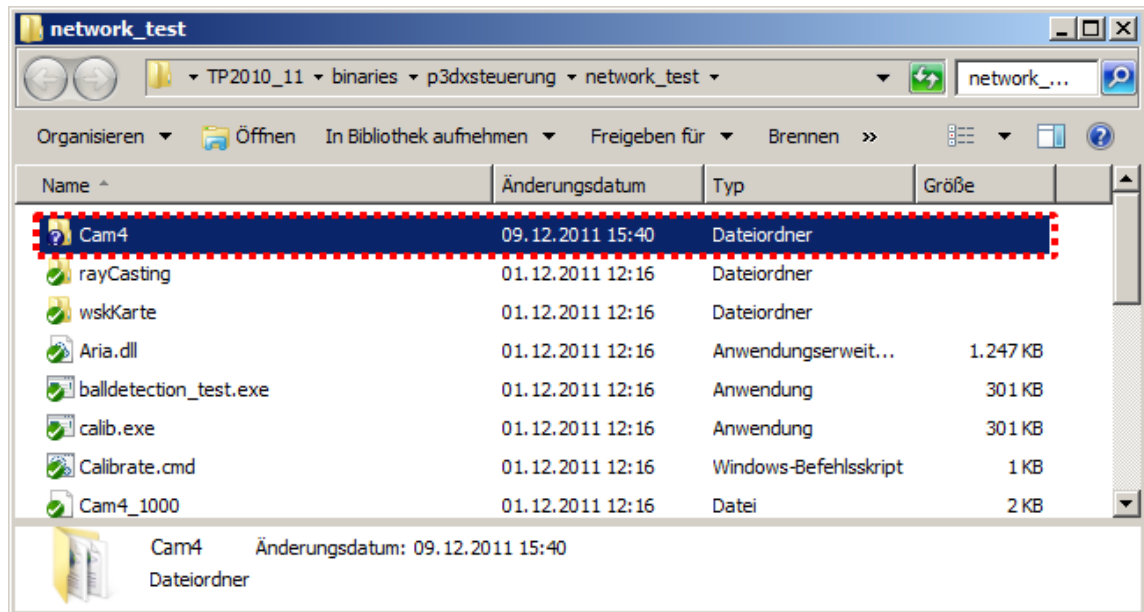
C:\Windows\system32\cmd.exe
[ -0.00654528  -0.11359  -0.993506  -399.363  1
[ 0  0  0  1  1

rTc
[ 0.999964  -0.00548719  -0.00654528  51.3611  1
[ -0.00619527  -0.993508  -0.11359  294.725  1
[ -0.00587951  0.113626  -0.993506  -436.009  1
[ 0  0  0  1  1

Kamera Hoehe: 294.725
Starte Kalibrierung
Chessboard Finder initialisiert
Chessboard groesse gesetzt
findchessboard aufgerufen
GetChessboard Pose
Bitte die Hoehe (in mm) des Schachbretts eingeben
Bo^CBatchvorgang abbrechen <J/N>? N

```

- Danach sollte ein neuer Ordner im aktuellen Verzeichnis mit den Kalibrierdaten sein. Er hat als Bezeichnung die Kameranummer erhalten. Im folgenden Bild sieht man das Ergebnis einer erfolgreichen Kalibrierung für die Kamera mit der Nummer 4, es wurde ein neues Verzeichnis `cam4` mit den Kalibrierungsdateien erzeugt:



#### 4.2.2 Einrichtung des Clients

Zunächst müssen die Webcam des Roboters und dessen serielle Schnittstelle mit dem Laptop, auf dem die Clientsoftware läuft, verbunden werden. Anschließend wird der Client über die Datei `config.cfg` eingerichtet. Eine Erklärung und ein Beispiel finden sich im Abschnitt 2.3.3 auf Seite 18. Die Datei muss sich im gleichen Verzeichnis wie die ausführbare Datei `p3dxSteuerung_threaded.exe` befinden. Ausserdem benötigt der Client ein Batchskript zum Starten, die Daten der Kalibrierung und Karten für die Lokalisierung. Idealerweise erzeugt man sich für die `*.exe`-Datei und anderen benötigten Dateien ein eigenes Verzeichnis, wo alle benötigten Dateien rein kopiert werden. Hierzu empfiehlt sich der in Abbildung 4.2.2 skizzierte Aufbau des Laufzeitverzeichnisses. Die Bedeutung der Elemente der Verzeichnisstruktur ist der Tabelle 4.1 zu entnehmen.

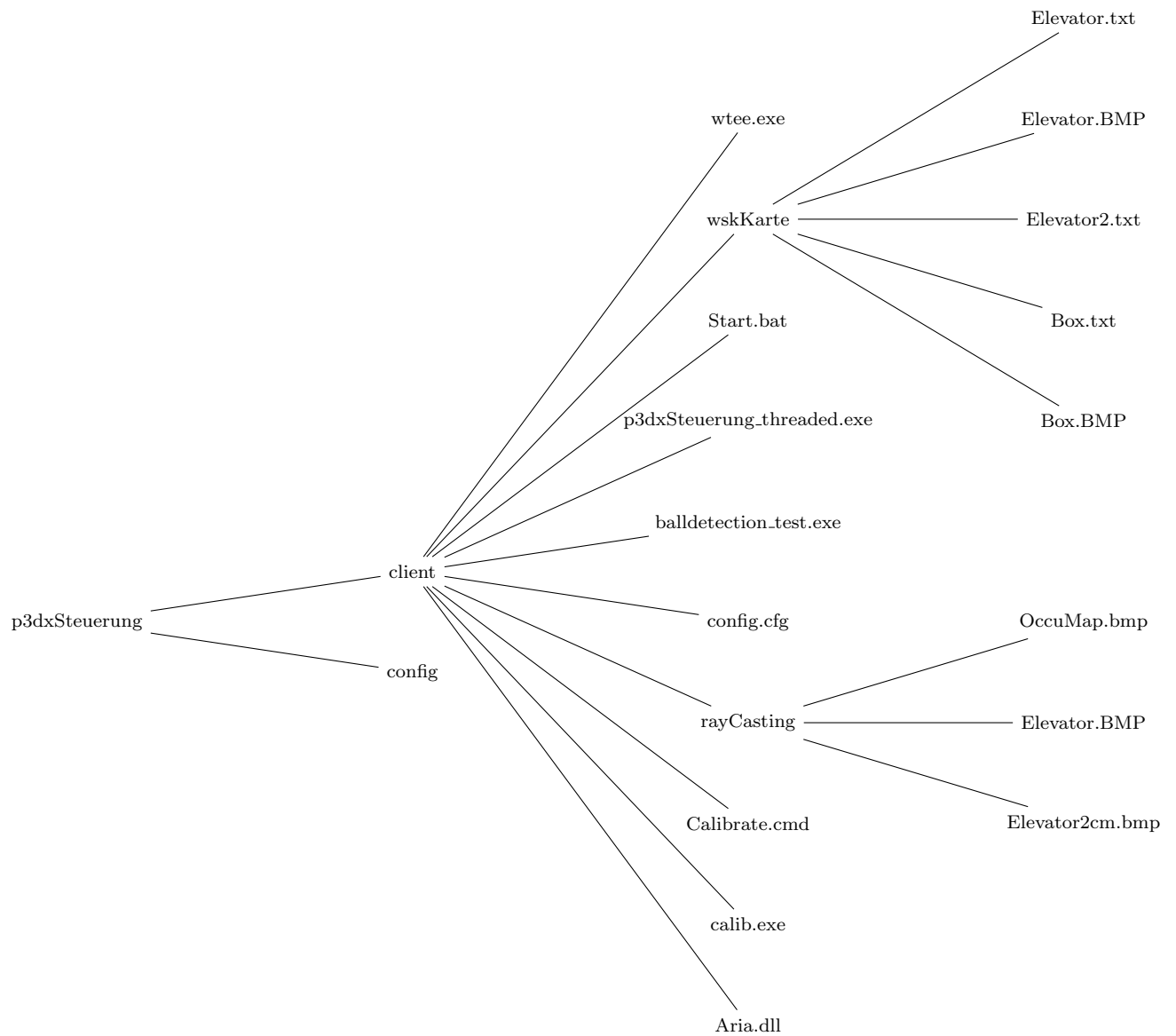


Abbildung 4.8: Empfohlener Aufbau des Laufzeitverzeichnisses



Name	Bedeutung
config	Vereichnis mit den *xml-Kalibrierungsdateien
client	Verzeichnis mit den Programmdateien des Clients
Aria.dll	DLLs, die vom Client zur Ansteuerung des Roboters
calib.exe	Hilfstool zur Kalibrierung der Ballerkennung <sup>1</sup>
Calibrate.cmd	Wrapper-Batchskript für die Kalibrierung
rayCasting, wskKarte	Verzeichnisse mit Raumkarten im BMP-Format
config.cfg	Konfigurationsdatei für den Client
balldetection_test.exe	Miniprogramm zum Testen der Ballerkennung <sup>2</sup>
wtree.exe	Windows-Tool, um Bildschirmausgaben von DOS-Programmen zu loggen und trotzdem am Bildschirm auszugeben. <sup>3</sup>
Start.bat	Batch-Skript, um den Client auszuführen und gleichzeitig ein Logging mittels wtree.exe zu ermöglichen.

Tabelle 4.1: Die Dateien des Laufzeitverzeichnisses und ihre Bedeutung

#### 4.2.3 Benutzung des Clients

Nach Konfiguration des Clients muss zunächst der Server gestartet werden, da der Client eine funktionierende Verbindung zum Server voraussetzt. Ist keine vorhanden, bricht er die Programmausführung ab. Nach Start des Servers positioniert man den Roboter in der Mitte des Raumes, in dem die Ballsuche stattfinden soll. Anschließend schaltet man den Roboter ein und startet den Client mit der Stapelverarbeitungsdatei **Start.bat**. Der Client startet nun und nimmt zunächst so viele Rotationen um die eigene Achse vor, wie in der **config.cfg** festgelegt worden sind. Diese dienen dazu, den Sonar-Partikelfilter zu initialisieren, sodass der Client dann die Position des Roboters im Raum bestimmen kann. Nach erfolgreicher Initialisierung verbindet sich der Client mit dem Server und wartet, bis dieser die Suche startet. Sobald vom Server die Suche gestartet wurde, wird der Client vom Server eine Liste von Punkten anfordern, die er dann abfahren wird. Gleichzeitig öffnet sich ein Fenster, das das aktuelle Bild der Kamera zeigt. Es wird während der gesamten Suche ständig aktualisiert.

Ab diesem Zeitpunkt muss man als Benutzer nur noch den Ablauf überwachen, falls es unerwarteterweise zu Kollisionen mit der Wand oder anderen Robotern kommt. Dies sollte zwar aufgrund der vom Server vorgenommenen Kollisionserkennung nicht passieren, allerdings lassen sich „False-Positives“ nicht generell ausschließen, mehr dazu im nächsten Kapitel. Erkennt der Client schließlich einen Ball im aktuellen Kamerabild, wird der Ball im Bild markiert:

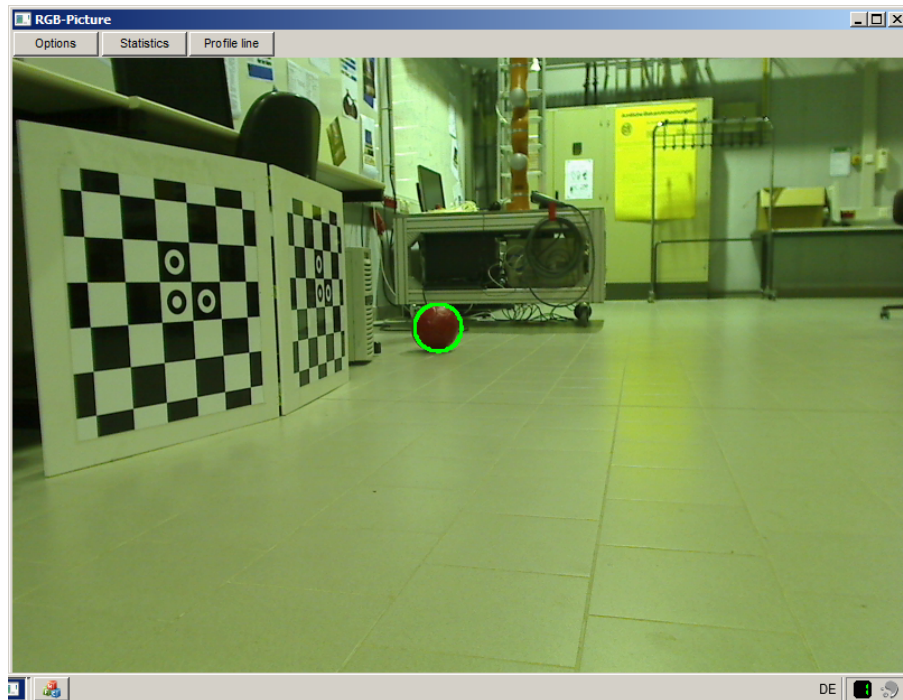


Abbildung 4.9: Kamerabild bei erfolgreicher Ballerkennung

Anschließend wird aus der mit dem Sonar-Partikelfilter ermittelten Position des Roboters im Raum und von der Ballerkennung zurückgegebenen Entfernung und Winkel vom Ball zum Roboter die Position des Balls im Raum bestimmt und an den Server übermittelt. Dieser weiß nun, dass die Ballsuche erfolgreich beendet ist und schickt allen verbundenen Clients eine Aufforderung die Suche zu beenden. In der Folge stoppen die Clients die Suche.

## 5 Evaluation

In diesen Abschnitt nehmen wir eine Bewertung der einzelnen Bestandteile der Software vor. Dabei geht es zum einen um die Client-Server Kommunikation, die Bahnplanung und Kollisionsvermeidung, und zum Anderen um die Lokalisierung des Clients im Raum mit dem Partikelfilter und die Ballerkennung.

### 5.1 Evaluierung des Servers

#### 5.1.1 Client-Server Kommunikation

Die zu testende Funktionalität ist das Übermitteln der Zielpunkte der geplanten Bahn an den Client und der Empfang der Roboterpositionen vom Client durch den Server. Bei unseren Experimenten traten keinerlei Probleme auf.

#### 5.1.2 Bahnplanung

Die Bahnplanung soll ja eine möglichst zufällige Auswahl von Punkten treffen, sodass die Roboter möglichst viele im Raum verteilte Positionen anfahren, sodass irgendwann der Roboter den Ball nahe genug kommt, um ihn zu erkennen. Dies leistet die Bahnplanung ohne Probleme.

#### 5.1.3 Kollisionsvermeidung

Hier ergab sich das Problem, dass wir bei unseren Versuchen nur bedingtes Kollisionspotential hatten, da der entsprechende Raum (kaum Problempotential hatte. Also wurde eine Version des Servers mit einer alternativen Karte gebaut. Diese Karte repräsentiert einen fiktiven Raum mit mehreren Zwischenwänden als mögliche Hindernisse. Natürlich liess sich diese Version aber nicht mit den realen Robotern testen, da der entsprechende Raum ja eigentlich nicht existiert. Also wurde auf den im Abschnitt 2.2.3 auf Seite 9 beschriebenen Testclient zurückgegriffen. Damit wurde eine Suche mit drei Robotern simuliert, die ohne Kollisionen verlief. Abbildung 5.1 zeigt die Simulation mit je einer Momentaufnahme für beide Kartenansichten. Danach haben wir getestet, ob die Erkennung der Wände auch mit einem Roboter im Raum vor dem Robotiklabor im Braunschweiger Informatikzentrum funktionieren würde. Wie oben erwähnt, hatte dieser deutlich geringeres Problempotential. Wie erwartet klappte dann auch die Vermeidung der Wände ohne Probleme. Nur einmal wurde eine Wand doch angefahren.

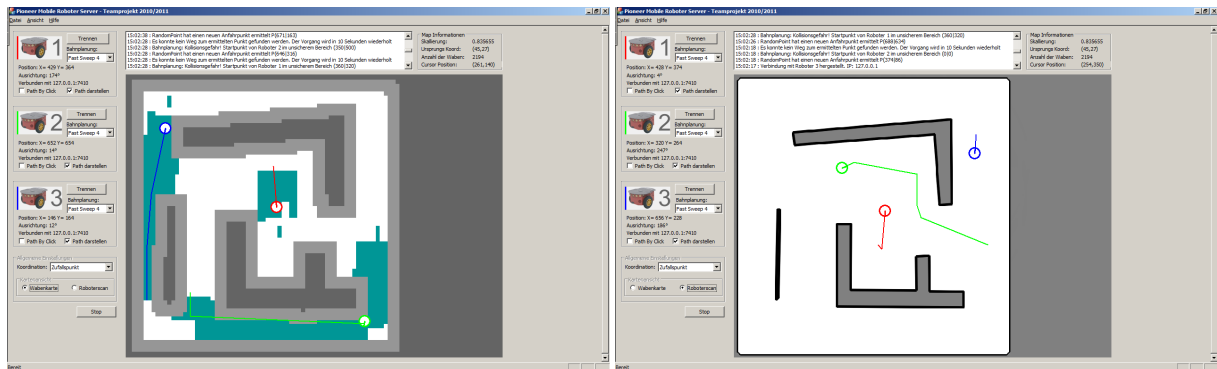


Abbildung 5.1: Waben- und Roboterscankarte beim Test der Kollisionserkennung

Dies war darauf zurückzuführen, dass die Positionserkennung mit dem Sonar-Partikelfilter kurzfristig einen Ausreißer hatte (vgl. hierzu auch den Abschnitt 5.2.1 ab Seite 38). Somit war aus Sicht der Kollisionsvermeidung keine Gefahr vorhanden und brauchte somit auch nicht abgewehrt werden. Im Endeffekt ist dieser „Fehlschlag“ also kein Zeichen, dass die Kollisionsvermeidung nicht funktionieren würde.

## 5.2 Evaluierung des Clients

### 5.2.1 Lokalisierung mit den Sonar-Partikelfilter

Bei unseren Versuchen die Position des Roboters im Raum zu bestimmen, stießen wir recht früh auf Probleme: So wurde regelmäßig eine komplett falsche Position im Raum bestimmt oder aber (gerade wenn der Roboter sich nahe einer Wand befand) die dem Roboter gegenüberliegende Position am anderen Ende des Raums als Position erkannt. Nach mehreren Versuchen stellten wir schließlich fest, dass die Lokalisierung besonders zuverlässig war, wenn der Roboter beim Start des Clients sich in der Mitte des Raums befand, sodass bei der Initialisierung des Partikelfilters durch die initiale Rotation der Abstand zu den Wänden zu beiden Seiten gleich war. Um nun der Ursache dieses Phänomens auf die Spur zu kommen, haben wir dann unser eigenes Visualisierungsskript geschrieben. Wir beschreiben nun zunächst die Funktionsweise und Installation des Skriptes, bevor wir uns den Ergebnissen zuwenden.

### 5.2.2 Visualisierung der Partikelmengen des Partikelfilters

Da die im Partikelfilter integrierte Visualisierung der Partikelmenge bei uns eine Zeit lang reproduzierbar zum Einfrieren unseres Clients geführt hat, haben wir den Partikelfilter, so erweitert, dass die Partikelmenge in eine Datei geschrieben werden kann ( 2.3.2) und passend dazu ein Skript (in Python unter Benutzung von pygame geschrieben), welches die Partikel anhand der Daten aus einer Partikelmeng-

gendatei über ein Bild legt. Dieses ist im Ordner Visualisierung im SVN-Repository unter dem Namen visualize.py zu finden. Wir beschreiben im Folgenden zunächst die Installation und Nutzung des Skriptes, bevor wir auf die Interpretation der Visualisierung eingehen

### Installieren der Abhängigkeiten des Visualisierungsskriptes und Benutzung des Skriptes

- Python 3.2 herunterladen und installieren<sup>1</sup>
- pygame 1.9.2a0 für Python 3.2 installieren<sup>2</sup>
- Python 3.2 zum PATH hinzufügen
- cmd/Eingabeaufforderung öffnen
- In den Ordner Visualisierung des Repositories wechseln
- Das Visualisierungsskript kann nun folgendermaßen benutzt werden:  
`python visualize.py (Dateiname des Bildes) (Dateiname der Partikelmengendatei)`  
z.b. `python visualize.py OccuMap.bmp visual.log`
- Das Ausgabebild hat dann den Namen V\_(Dateiname der Partikelmengendatei)\_(Dateiname des Bildes)

### Interpretation der Visualisierung

In den folgenden Bildern sieht man jeweils die Partikelverteilung, wie sie sich nach der initialen 360°-Rotation zur Partikelfilter-Initialisierung darstellt. In den oberen beiden Bildern wurde dabei der Roboter an der linken respektive rechten Wand des Raums gestartet, in den unteren beiden in der linken respektive rechten Hälfte der Raummitte:

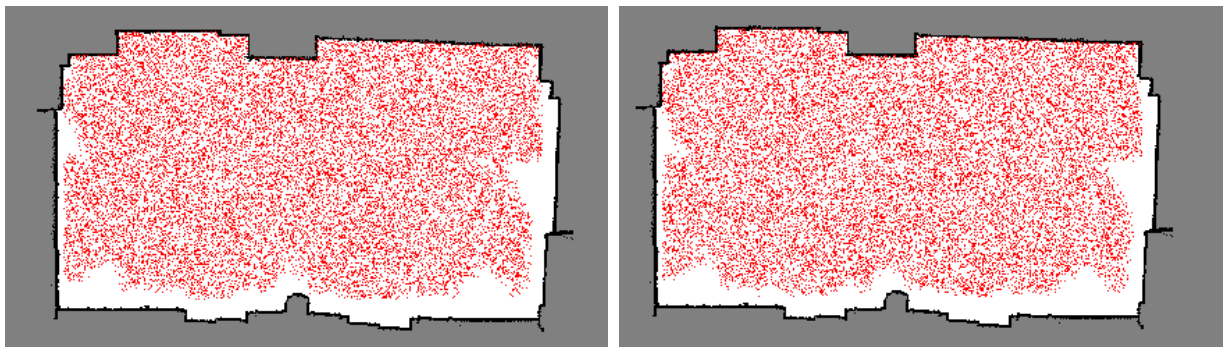


Abbildung 5.2: Partikelverteilung im Partikelfilter bei Start nahe der Wände

<sup>1</sup><http://python.org/ftp/python/3.2.2/python-3.2.2.msi>

<sup>2</sup><http://pygame.org/ftp/pygame-1.9.2a0.win32-py3.2.msi>

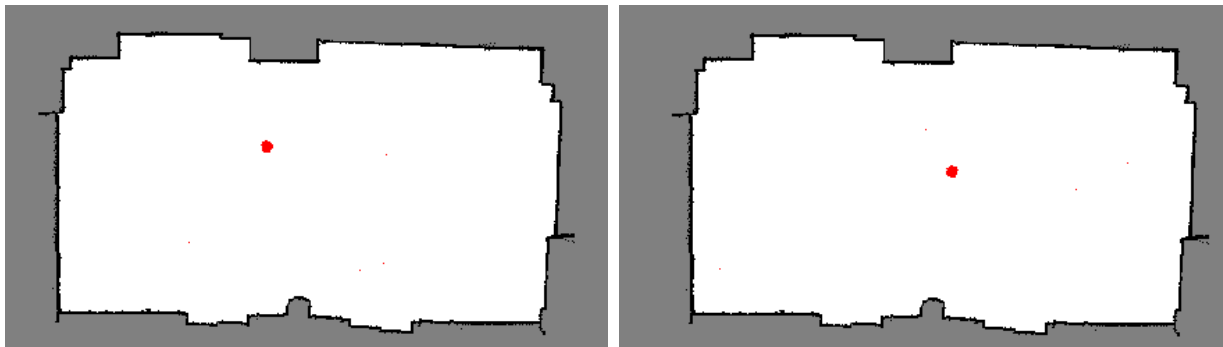


Abbildung 5.3: Partikelverteilung im Partikelfilter bei Start in der Raummitte

In den oberen beiden Bildern sind die Partikel auf der gesamten Raumkarte verteilt, somit ist es unmöglich aus einer Partikelhäufung auf eine bestimmte Position des Roboters im Raum zu schließen. In der Tat entsprach bei unseren Versuchen die vom Client an den Server übergebene Position nur in seltenen Fällen der Tatsächlichen, wenn wir wie in den beiden dargestellten Fällen, den Client an den Wänden des Raumes gestartet haben. Haben wir hingegen, wie in den unteren beiden Fällen, eine Position nahe der Raummitte gewählt, erzielten wir im Allgemeinen sehr gute Ergebnisse für die initiale Position des Roboters auf der Karte. Sie entsprach meistens der Tatsächlichen. Dies ist wenig überraschend, wenn man die Visualisierung betrachtet: Es ist eine klare Konzentration der Partikel auf die Startposition des Roboters zu beachten.

Ob im weiteren Verlauf der Ballsuche brauchbare Ergebnisse bei der Lokalisierung kamen, war analog stark von der initialen Lokalisierung abhängig: War schon die Startposition falsch, änderte sich dies auch nicht im Verlauf der Ballsuche. War sie hingegen richtig, wurden in der Folge auch die weiteren Positionen korrekt bestimmt. Trotzdem kam es mitunter, insbesondere wenn sich der Roboter in der Nähe von Raumecken oder Wänden aufhielt zu Sprüngen. Die vorhandenen Sprünge sind in einen akzeptablen Rahmen, der die Funktionsweise des Clients nicht gefährdet.

### 5.2.3 False-Positives bei der Ballerkennung

Wie bereits im Abschnitt 2.3.1 erwähnt, kann es bei der Ballerkennung zu False-Positives kommen. Diese liegen in der Funktionsweise der Ballerkennung begründet: Nach Aufnahme des Kamerabildes wird das Bild vom RGB- in den HSV-Farbraum umgewandelt, der anschließend nach roten Partikeln gefiltert wird. Erreichen diese einen gewissen Schwellwert, werden diese nach kreisförmigen Objekten durchsucht. Anschließend wird die Wahrscheinlichkeit berechnet, ob es sich dabei um einen Ball handelt. Wenn nun aber im Kamerabild so ein rotes Objekt ist (z.B: Ein anderer p3dx-Roboter), wird dieses mit hoher Wahrscheinlichkeit als „Ball“ erkannt. In unseren Versuchen wurden unter anderen der Feuerlöscher des Labors, aber auch die roten Schränke gerne als „Bälle“ erkannt. Dies konnten wir durch Aufbau

eines geeigneten Sichtschutzes verhindern, allerdings kam es immer noch zu False-Positives an Stellen, wo weder ein rotes Objekt, noch sonst irgendein Objekt vorhanden war. Wir vermuten, dass dann höchstwahrscheinlich im Licht der den Vorraum des Fahrstuhl vor dem Robotiklabor beleuchtenden Lampen so viele Rot-Partikel vorhanden sind, dass auch da unter Umständen False-Positives auftauchen können.

## 6 Zusammenfassung

Unsere Aufgabenstellung eine koordinierte Ballsuche zu implementieren, ist uns im Wesentlichen gelungen:

Ein zentraler Server nimmt die Bahnplanung vor und ist in der Lage die Bahnplanung für bis zu drei Clients vorzunehmen. Diese Bahn wird von der Clientsoftware ohne größere Probleme abgefahren. Mit der Hilfe der uns zur Verfügung stehenden Ballerkennung und des auf einen früheren Teamprojekt aufbauenden Sonar-Partikelfilters war es möglich, den Client so zu entwickeln, dass er nicht nur seine eigene Position im Raum, sondern auch eventuell dort vorhandene Bälle bestimmen kann.

Zwar erwies sich der Partikelfilter in bestimmten Situationen als nur bedingt zuverlässig, diese Problematik lässt sich aber durch eine geschickte Wahl der Initialisierungsposition umgehen. Die Ungenauigkeiten lassen sich im Endeffekt darauf zurückzuführen, dass der dafür genutzte Ultraschall zwar schon im Roboter vorhanden, aber nicht gerade der beste denkbare Ansatz ist. Theoretisch wäre es auch denkbar, die Roboter zusätzlich zum eingebauten Sonar mit einem Laserscanner auszustatten, dieser sollte vom Ansatz her deutlich zuverlässigere Ergebnisse liefern. Eine Implementierung dieses Ansatzes lag allerdings nicht im Fokus unseres Projektes.

Grundsätzlich kann also das Projekt als Erfolg betrachtet werden.



## Literaturverzeichnis

- [1] Tobias Breuer. Kamerabasierte Ballerkennung und Geschwindigkeitsschätzung. Bachelorarbeit., Institut für Robotik und Prozessinformatik. Technische Universität Braunschweig 25. Mai 2011.
- [2] P. E. Hart, N. J. Nilsson, and B. Raphael: IE. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, 1968.
- [3] MobileRobots Inc. *Pioneer 3 Operations Manual with MobileRobots Exclusive Advanced Robot Control & Operations Software*, Januar 2006. [http://www.ist.tugraz.at/\\_attach/Publish/Kmr06/pioneer-robot.pdf](http://www.ist.tugraz.at/_attach/Publish/Kmr06/pioneer-robot.pdf).
- [4] Pete Shinnars. *Python Pygame Introduction*, 2011. <http://www.pygame.org/docs/tut/intro/intro.html> [Online; Stand 18. Dezember 2011].
- [5] Wikipedia. A\*-Algorithmus — Wikipedia, Die freie Enzyklopädie, 2011. [http://de.wikipedia.org/w/index.php?title=A\\*-Algorithmus&oldid=96923702](http://de.wikipedia.org/w/index.php?title=A*-Algorithmus&oldid=96923702) [Online; Stand 18. Dezember 2011].