



Софийски университет „Св. Кл. Охридски”

Факултет по математика и информатика

*Катедра „Компютърна информатика”*

# ДИПЛОМНА РАБОТА

на тема

„Разработване на модул за откриване и  
заобикаляне на препятствия с  
хуманоиден робот Nao”

Дипломант: **Антон Красимиров Чолаков**

Магистърска програма: **Изкуствен интелект**

Факултетен номер: **M23295**

Научен ръководител:  
**проф. д-р Мария Нишева**

София, 2015 г.

**Резюме:**

Хуманоидните роботи имат много възможни употреби в човешкия живот и работа. Поради тяхното физическо подобие на хора, те могат да бъдат добре приети от тях и способни да обхождат и използват заобикалящата среда, която отново е предназначена за хора. Въпреки това, една такава среда е по-трудна за моделиране и създаването на прости процедури за навигация на робот в нея. Затова е желателно роботът да бъде способен автономно да засича и заобикаля препятствия по пътя си. Един възможен подход за решаване на този проблем е компютърно зрение чрез употреба на камера. Този проект цели да имплементира един такъв модул за хуманоидния робот Nao.

**Декларация за авторство:**

Аз, Антон Чолаков, с настоящото декларирам, че следното е резултат на моя личен труд под ръководството на проф. Мария Нишева. Всички използвани източници са цитирани в секцията за използвана литература. Всички библиотеки и фрагменти от код, които са чужда разработка, са цитирани в секцията за използвана литература.

# Съдържание

<b>Глава 1. Увод.....</b>	<b>4</b>
1.1. Актуалност на проблема и мотивация.....	4
1.2. Цел и задачи на дипломната работа.....	5
1.3. Очаквани ползи от реализацията.....	5
1.4. Структура на дипломната работа.....	6
<b>Глава 2. Преглед на предметната област.....</b>	<b>7</b>
2.1. Основни понятия.....	7
2.2. Подходи и методи за заобикаляне на препятствия от робот.....	8
2.3. Съществуващи решения.....	9
2.4. Изводи.....	11
<b>Глава 3. Използвани модули и функционалност на робота Nao.....</b>	<b>12</b>
3.1. Платформа за създаване на модули за робота Nao.....	13
3.2. Модул за памет и събития.....	15
3.3. Модул за движение.....	15
3.4. Модул Визуален компас.....	16
3.5. Модул сонар.....	16
<b>Глава 4. Анализ на задачата.....</b>	<b>17</b>
4.1. Концептуален модел.....	17
4.1.1. Опростяване на задачата.....	17
4.1.2. Желан потребителски интерфейс.....	18
4.2. Откриване на препятствия чрез визуалния компас.....	18
4.2.1. Работа и резултати на визуалния компас.....	19
4.2.2. Определяне на разстояние до обект.....	20
4.3. Заобикаляне на препятствия.....	24
4.3.1. Определяне на блокиращи препятствия.....	24
4.3.2. Определяне на междинна позиция.....	25
4.4. Изводи.....	28
<b>Глава 5. Проектиране.....</b>	<b>29</b>
5.1. Обща архитектура.....	29

<u>5.2. Команден слой.....</u>	31
<u>5.3. Слой, задаващ целеви пози.....</u>	32
<u>5.4. Слой, извършващ движенията.....</u>	33
<u>5.4.1. Компонент за откриване на препятствия.....</u>	34
<u>5.4.2. Компонент за заобикаляне на препятствия.....</u>	35
<u>5.4.3. Компонент, използващ сонара.....</u>	35
<b>Глава 6. Реализация и тестване.....</b>	<b>36</b>
<u>6.1. Реализация на слоевете и компонентите.....</u>	36
<u>6.1.1. Потребителски интерфейс.....</u>	36
<u>6.1.2. Команден слой.....</u>	37
<u>6.1.3. Метод за извлечане на текущата поза на робота.....</u>	38
<u>6.1.4. Слой, задаващ целеви пози.....</u>	39
<u>6.1.5. Слой, извършващ движенията.....</u>	40
<u>6.1.6. Компонент за откриване на препятствия.....</u>	44
<u>6.1.7. Компонент за заобикаляне на препятствия.....</u>	49
<u>6.1.8. Компонент, използващ сонара.....</u>	52
<u>6.2. Интеграция.....</u>	53
<u>6.2.1. Инсталриране и използване.....</u>	53
<u>6.2.2. Интеграция със средата за програмиране Choregraphe.....</u>	54
<u>6.3. Тестване.....</u>	55
<u>6.4. Анализ на резултатите от тестването.....</u>	57
<b>Глава 7. Заключение.....</b>	<b>58</b>
<u>7.1. Обобщение на изпълнението на началните цели и претенциите за оригинални резултати.....</u>	58
<u>7.2. Насоки за бъдещо развитие и усъвършенстване.....</u>	58
<b>Използвана литература.....</b>	<b>60</b>
<b>Фигури, таблици и снимки.....</b>	<b>62</b>

# Глава 1. Увод

## 1.1. Актуалност на проблема и мотивация

Роботиката е актуална тема в наши дни, защото се търси все повече задачи да бъдат автоматизирани. По този начин човешкият труд може да бъде заменен от извършвана от машина работа. Това може само по себе си да е целта – продажба на автоматизирани прахосмукачки, които да облекчават домакинската работа, или безпилотни военни машини, непоставящи в опасност живота и здравето на войници. Автоматизацията на работата може и да бъде с цел намаляване на производствени разходи и повишаване качеството на продукцията. Съществуват и множество други причини за създаването на роботи и тепърва все повече машини ще взимат активно участие в човешкото ежедневие с цел забавление, помощ, обучение и т.н.

Способността на един робот да се движи неимоверно много би увеличила неговата полезност, защото ще позволи роботът да е там, където е необходим, а не просто там, където е монтиран. Още по-добре би било, ако роботът е способен да се придвижва в произволна среда или поне такава, която задава определени правила, но не е напълно известна предварително. Именно в тази посока са насочени множество съвременни изследвания, защото решаването на тази задача ще отвори много нови възможности пред роботите и тяхната употреба.



Снимка 1: Хуманоиден робот Nao

Автономното откриване и заобикаляне на препятствия при движение е важна стъпка при самостоятелното навигиране на робот в не напълно известна среда. Това важи в пълна сила за хуманоидния робот Nao (на Снимка 1), чиято идея е да бъде интегриран в ежедневието на хората, за развлечение и/или изпълнение на задачи, понеже в човешките домове и офиси често се поставят предмети на по-рано свободни места. Това означава, че ако навигацията се извършва без постоянно наблюдение над средата, в която се извършва движението, роботът може да се сблъска с някой предмет.

За справяне с тази задача съществуват различни подходи, един от които е компютърното зрение (*Computer Vision*). Именно по този начин настоящата дипломна работа ще се опита да разреши този проблем. Използвайки вградената камера на робота Nao, препятствията на пътя му могат да бъдат открити и да се намери друг маршрут, който да ги заобиколи, след което роботът отново да се опита да достигне първоначално зададената си цел.

В допълнение на камерата, вграденият сонар също може да бъде използван, но поради ограничения му обхват, той би служил като обезопасителен метод, в случай че компютърното зрение пропусне обекти на пътя на робота. При засичането на обекти, той ще сигнализира на робота да прекрати движението си.

## **1.2. Цел и задачи на дипломната работа**

Целта на дипломната работа е да се разработи модул за управление движението на робота Nao, който при зададена дестинация да открие и заобиколи евентуалните препятствия, които се окажат на неговия път, позволяйки му да достигне крайната си точка без да влиза в контакт с други обекти.

За да се постигне това, трябва първо да се проучи платформата за програмиране на робота Nao и да се анализират неговите хардуерни и софтуерни възможности. След това предстои да се разгледат методи за откриване на препятствия и да се избере такъв, подходящ за разработване и употреба от робота.

След като всичко бъде планирано, предстои модулът да бъде реализиран и интегриран с платформата на робота, след което той трябва да бъде тестван и резултатите анализирани.

Като последна задача на дипломната работа ще бъде реализиран и начин за интегриране на готовия модул със стандартната среда за програмиране на робота Nao – Choreograph.

## **1.3. Очаквани ползи от реализациите**

Ползите от реализациите на модула са няколко. Първата е, че ще се разгледа и изпробва един от методите за откриване и заобикаляне на препятствия. Това ще даде информация относно това дали този метод е теоретично издържан и използваем. Втората полза е, че ще се разгледат и анализират начините за работа с робота Nao и интеграцията с него. Трето, че се

провери дали избраният метод за откриване и заобикаляне на препятствия е осъществим въз основа платформата на робота Nao.

Накрая, ако реализацията на модула е успешна, то роботът, като платформа за програмиране, ще се сдобие с функциониращ модул за придвижване, който ще открива и заобикаля препятствия – нещо, с което той все още не разполага. Това ще позволи на други програми ефективно да използват този модул в своята логика от по-високо ниво. Пример за такава логика е модул, който задава на робота да се придвижи от една известна локация в дома или офиса до друга такава, например от хола до кухнята. Тук локациите „хол“ и „кухня“ са предварително известни като местоположение и командващият модул задава на модула за движение серия от команди за това къде трябва да отиде роботът. И тук, ако за модул за движение се използва разработения за дипломната работа модул за откриване и заобикаляне на препятствия, то роботът успешно ще избегне неочекваните препятствия по пътя си от хола до кухнята.

## **1.4. Структура на дипломната работа**

Дипломната работа се състои от следните седем глави:

- Първа глава представлява увод – описание на проблема и поставените цели и задачи.
- Втора глава въвежда читателя в предметната област, изяснявайки някои общи понятия. Също така, в нея са разгледани подходи и методи за заобикаляне на препятствия от робот, включително съществуващи проучвания и разработки. Накрая са направени някои изводи.
- Трета глава представя робота Nao и платформата за разработване на модули за него. Описани са още модулите, които дипломната работа използва за постигане на целите си.
- В четвърта глава е направен анализ на поставената задача и са представени алгоритмите, които ще бъдат използвани в реализацията на разработвания модул.
- Пета глава се занимава с проектирането на модула – потребителски интерфейс, разделение по слоеве и компоненти, както и логиката на работа на всеки един от тях.
- Шеста глава разглежда направената реализация и осъществените тестове на модула. В нея е извършен и анализ на резултатите и срещнатите проблеми.
- Глава седем прави финални заключения за направеното в тази дипломна работа и представя насоки за бъдещо развитие и усъвършенстване и разработения модул.

Накрая на дипломната работа има списък с използваната литература, последван от списъци на фигураните, таблиците и снимките в дипломната работа.

# Глава 2. Преглед на предметната област

## 2.1. Основни понятия

*Робот* е изкуствено създаден механичен или виртуален (софтуерен) агент, способен да извърши някаква дейност автономно или полуавтономно. Физическият робот може да бъде свободно подвижен или монтиран някъде, например поточна линия, с хуманоидна или друга форма. Виртуалният робот, обикновено наричан само бот, е нематериален и обикновено представлява само софтуер управляващ нещо. Предназначението на един робот може да бъде практично, образователно, развлекателно или друго, като е възможно множествено предназначение. Работите с практична цел могат да управляват машина, включително софтуер, която принципно е предназначена за човек, или да извършват сами дадена работа, която иначе би била досадна, опасна или просто по-скъпа, ако се извърши от човек. Примери за такива роботи са съответно автопилота на самолети или круз-контрол (*cruise control*) на автомобили и автоматични прахосмукачки или асембилиращи роботи на поточна линия. Развлекателните роботи служат за разтоварване, упражнения или друга дейност, която би доставила забавления за хората. Такива са например софтуерът управляващ агенти в компютърни и видео игри или машина хвърляща топки за бейзбол или тенис.

*Хуманоиден робот* е робот, който наподобява човек. Подобието може да бъде физическо, поведенческо или и двете. Работите наподобяващи физически човек се приемат по-добре от хората и могат да бъдат по-лесно пригодими за работа в човешка среда. Работите, чието поведение наподобява човешко, могат да бъдат партньори в разговори или игри, макар в общия случай те да не са все още на необходимото ниво на развитие.

*Среда* на робота ще наричаме всичко, което го заобикаля – терен или помещение, климатични и светлинни условия, обекти, други агенти и т.н. Работът може да си взаимодейства със средата, възприемайки я и/или въздействайки ѝ. Примери за това са регистриране на движение или звукови команди и преместване на предмети или дори самия себе си. Това понятие е изключително общо и обхващащо.

*Статична и динамична* среда на робота са две взаимноизключващи се понятия. Първото означава, че средата на робота не се променя от само себе си, т.е. тя е статична. Това теоретично е невъзможно, но на практика е достатъчно най-важните елементи на средата да не се променят. Пример за такава среда е затворена стая с постоянно осветление, в която по време на работа на робота нищо не се движи. Динамична среда е противоположното твърдение – средата на робота се променя независимо от него и неговите действия. Пример за такава среда е човешки дом или офис, из който често или постоянно се движат хора. Важно е да се уточни, че промяната в една статична среда в следствие на действията на робота, по един предвидим и логичен начин, не я прави динамична. Работата в динамична среда е много по-сложна, защото тя трудно се

поддава на моделиране и един робот, опериращ в такава среда, трябва да е готов и способен да реагира на много различни и непредвидени ситуации. Поради тази причина, поне за момента, в повечето случаи в сферата на роботиката се разглежда статична среда.

*Движение* на робота в средата е промяната на неговата позиция в пространството. Това може да варира от промяна в положението на част на робота, като например повдигане на крайник, до промяна в местоположението, като например придвижване от единия край на помещение до другия. Възможността за движение на един физически робот е изключително важна, защото в противен случай той би бил ограничен единствено до звукови или светлинни сигнали, което значително намалява неговата полезност.

*Препятствие* е обект, който би могъл да ограничи движението на робот. Това обикновено са различни предмети, които стоят на пътя на робота и или му пречат да премине, или биха били въздействани по някакъв начин от колизията си с него. Пример за двата вида са стени или тежки мебели, през които роботът не може да премине и не може да отмести, и различни леки предмети, които биха били деформирани или избутани, като хартиена кутийка или топка. Обикновено препятствията трябва да бъдат заобиколени или превъзмогнати по друг начин, в зависимост от тяхното естество, средата, възможностите и предочитаното поведение на робота.

## **2.2. Подходи и методи за заобикаляне на препятствия от робот**

Обикновено при възлагане на дадена задача на робот, включваща придвижване из неговата среда, той разполага с някаква форма на познание за тази среда. В повечето случаи това е карта на помещението, така че стените и неподвижната част от мебелите са известни. Проблем обаче са подвижните мебели и най-различни по-малки предмети, които също могат да създават трудности при движението на един робот. Възможно е и да няма налична карта на средата, в който случай или роботът трябва да състави сам такава, или да работи без карта, стига естеството на работата му да го позволява.

Най-простият подход за заобикаляне на препятствия е, ако роботът разполага с тяхното местоположение. Това означава, че те са част от картата на средата, с която разполага роботът. В този случай задачата се свежда до намиране на път до целта на движението. Проблемът тук е, че рядко се знае къде са препятствията. Това би означавало, че при всяка промяна на обстановката картата на робота ще трябва да се обновява изрично и докато това понякога е възможно, рядко ще е практично. В условията на поставената от дипломната работа задача, такава карта не е налична.

Друг много прост подход е роботът да открива препятствията си сам, като се удря в тях. Възможно е това да става сравнително безопасно чрез броня с амортизори. По този начин роботът изследва сам средата си и или построява карта, или просто открива препятствията по пътя си, и се опитва да ги заобиколи. Този принцип на работа обаче е подходящ само за някои видове роботи и някои видове околнни среди. За хуманоидния робот Nao, който е

сравнително лек и нестабилен, но пък достатъчно тежък да смачка малки и крехки предмети, този начин на работа не е подходящ.

Следващият подход за решаване на задачата е чрез употреба на измервателни уреди като инфрачервени или лазерни сензори или сонар. При тях роботът активно изпраща сигнал, инфрачервен или лазерен лъч или ултразвукова вълна, и отчита дали, кога и къде ще приеме обратно този сигнал. Лъчът или звуковата вълна ще се отразят или отскочат от препятствието под определен ъгъл и, в зависимост от разстоянието до обекта, ще се получи в различен момент. По този начин роботът научава за наличието на препятствие и посоката и разстоянието до него. Сред проблемите при употребата на сонар е, че е сравнително неточен в по-сложна среда. Също така при робота Nao, обсегът на сонара е (софтуерно) ограничен до половин метър<sup>[1]</sup>, което е недостатъчно за целите на дипломната работа. Инфрачервените и лазерните уреди за измерване на разстоянията са по-точни, но и по-скъпи. По-важното е, че роботът Nao няма вградени такива и за да разполага с тях е необходимо да се направи екземпляр по поръчка.

Възможен подход е и употребата на (компютърно) зрение чрез камера. Снимайки околната среда, един или множество пъти, е възможно да се извлече информация за нея. Животните, включително хората, използват зрение, за да си създават представа за заобикалящия ги свят. При тях двумерните изображения са проекциите на света върху ретините на очите им.

Чрез употребата на две или повече камери, които снимат едновременно от различни известни позиции, се получава стерео зрение (*stereo vision*)<sup>[2]</sup>. То се използва от множество животни или машини, защото позволява възприемането на дълбочина на изображенията и по този начин изграждането на триизмерни модели на околната среда. В случая на робота Nao обаче стерео зрение не може да се използва, понеже двете му камери имат много малко припокриващо се зрително поле.

Постигането на триизмерно зрение обаче е възможно и само с една камера, която снима от различни позиции, отчитайки движението, което камерата е извършила. Това се нарича структура от движение (*structure from motion*)<sup>[3]</sup> и то се използва от животните, заедно със стерео зрението. Структура от движение е техника за извличане на триизмерната структура на обекти или среда от набор от двумерни изображения. Това става, като особени точки се проследяват между две или повече двумерни изображения и в зависимост от известването им, отчитайки и промяната в разположението на камерата, се изгражда триизмерен модел на наблюдаваното. За да се осъществи това, трябва да има някакво движение – или на самите обекти, или на камерата. В частност паралакс метод е начин за анализиране на изображенията и извличане на разстоянията до точки в тях, което отговаря на дълбочината на образа. Разполагайки с камера и движейки се, за робота Nao това е един постижим подход за откриване на препятствия в околната среда.

## 2.3. Съществуващи решения

За заобикаляне на препятствия по време на работа, автономните прахосмукачки Roomba® на фирмата iRobot® използват подхода с леки удари в препятствията. Роботът е снабден с броня и амортизори и чрез два датчика за

натиск разбира къде се намира обектът, в който се е ударил. Решението е просто, ефективно и практически, но както се вижда от видеото<sup>[4]</sup> не е на нужното по-елегантно ниво, което се очаква от един хуманоиден робот като Nao.

Важно е да се отбележи, че има редица материали и опити да се имплементира локализация с робота Nao, подобрявайки одометрията на робота с визуални данни, лазерни датчици за разстояние или камери, отчитащи дълбочина в изображенията. Kooijman и други разработват система NAVIGATE, работеща подобно на модула визуален компас на робота Nao – извличат се ключови точки от поредица изображения, напасват се и промените се анализират.<sup>[5]</sup> Системата им обаче не е в работещо състояние. Fojtu и други работят в същата насока, като тяхната имплементация е успешно тествана в коридора на офиса им, но изисква предварително роботът да бъде разведен ръчно.<sup>[6]</sup> George и Mazel от Aldebaran Robotics използват предварително залепени из помещението бар кодове, получавайки добри резултати<sup>[7]</sup>. Wei, Xu и други използват стандартните за домашна обстановка забележителни обекти, но разчитат на предварително зададена карта на мястото с отбелязани на нея забележителности.<sup>[8]</sup> Любопитно е, че някои от описаните методи работят по същия начин като визуалния компас на фирмата производител на робота Aldebaran<sup>[12]</sup>, но не го използват.

Съществуват няколко разработки за навигиране на робот в среда с препятствия, използвайки лазери или камери, отчитащи дълбочина. Hornung и други построяват забележителна система, прилагаща Монте Карло алгоритми и използваща лазер или дълбочинна камера, обикновена камера, датчиците на ставите на робота и други данни. Резултатната система определя 6-мерна поза на робота в пространството и демонстрира забележителни резултати свързани с локализация, коригиране на движението, отчитане на препятствията и други, включително способност да се изкачват стълби. Тя обаче разчита на предварително предоставена карта на заобикалящата среда, като планират в бъдеща работа да използват алгоритъм за едновременно картографиране и локализиране (*Simultaneous Localization and Mapping (SLAM)*), за да елиминират тази нужда.<sup>[9]</sup> Maier и други разработват метод за заобикаляне на препятствия, използващ камерата на робота Nao и редки измервания с лазер, трениращ класификатори разпознаващи проходими места и непроходими препятствия. В резултат се построява и постоянно обновява двумерна карта на равнината, в която се разграничават местата, откъдето роботът може да премине.<sup>[10]</sup>

Опитите за справяне със задачата за заобикаляне на препятствия само с обикновената камера на робота Nao са по-малко. Havlena и други правят опит за изграждане на триизмерен модел на заобикалящата среда, използвайки само стандартните камери на робота. Срещат обаче проблеми с малкото зрително поле на робота. Също така, за да получат две или повече изображения от различни позиции, възприемат подход на кратки движения напред, спиране, оглеждане и клякане.<sup>[11]</sup> Това прави движението на робота изключително бавно. Построяването на триизмерен обект ще позволи откриването на препятствията, но работата им все още не е стигнала този етап.

## **2.4. Изводи**

Разгледаните материали показват, че са проучвани различни начини за решаване на задачата за откриване и заобикаляне на препятствия и близко свързаната задача за локализация на робота и коригиране на движенията му. Въпреки това, само една от намерените разработки се опитва да атакува проблема, използвайки само наличните уреди в стандартен модел робот Nao, като работата ѝ изисква чести спирания и сравнително тежки изчисления, което е нежелано.

И макар да има постигнати много добри резултати, когато роботът разполага с лазер или камера, измерваща дълбочина в изображенията, това оборудване е по-скъпо и недостъпно. Затова има стойност в опита да се реши задачата, използвайки само обикновен робот Nao.

Още повече, сред намерените и разгледани работи няма опит да се използва структура от движение, използвайки движението на робота напред към целта. Също така, както бе споменато по-горе, употребата на модула визуален компас не е разгледана. Тези фактори означават, че има смисъл да се направи опит за решаване на задачата за откриване и заобикаляне на препятствия по представения в тази дипломна работа начин.

## Глава 3. Използвани модули и функционалност на робота Nao

За разработването на модула на дипломната работа на разположение бяха два робота Nao, версия V4, любезно предоставени от ръководството на Центъра за Източна Европа на Европейския софтуерен институт (на Снимка 2). Освен това те дадоха възможността за работа по дипломната работа в техния офис, намиращ се в София, България, и помагаха с някои идеи, разговори и при възникването на хардуерни проблеми с роботите. За всичко това авторът на настоящата дипломна работа им е дълбоко благодарен!



Снимка 2: Nao роботите ESSYIKA (оранжев) и ESSYIKO (син)

Роботът Nao (V4) представлява хуманоид с височина 57,3 сантиметра, ширина 27,5 сантиметра и дължина с изпънати ръце 31,1 сантиметра, тегло 4,3 килограма и 25 степени на свобода<sup>[13][14]</sup>. В главата на робота се намира процесор Intel Atom с мощност 1,6 GHz и 1 GB оперативна памет<sup>[13]</sup>. В торса му пък се намира литиева батерия, която осигурява около 90 минути автономия<sup>[13]</sup>, като този интервал варира в зависимост от употребата на робота и износването на самата батерия.

Връзката с робота се осъществява посредством локална мрежа – по кабел или безжично. Първото се използва по-скоро за първоначална конфигурация, след което е нормално да се използва безжична връзка, позволявайки на робота да се движи свободно.

Nao разполага с две камери с висока резолюция, четири микрофона, сонар, жироскопи и акселерометри и други датчици. Важно е да се уточни, че двете камери са разположени в главата една под друга, в средата на челото и на мястото на устата, като те имат много малко припокриващо се зрително поле.

Софтуерът работещ на роботите бе операционната система Naoqi 2.1.3.3, като тази версия трябва да е идентична с версията на комплекта за разработване на софтуер (Software Development Kit) за робота<sup>[24]</sup> и средата за програмиране на логика от по-високо ниво, Choregraphe. Всичко това бе свалено от сайта на Алдебаран<sup>[15]</sup> през акаунт, отново предоставен от Европейския софтуерен институт.

За компилация е използван инструмента qBuild, който от своя страна работи със CMake – крос-платформена система за компилация.<sup>[24]</sup>

Единственият избор за използвани инструменти е този на операционна система и текстов редактор. Това е така, понеже средата за програмиране на робота Nao е строго фиксирана. Възможно е кодът да се пише на три различни операционни системи – Linux, Windows, OS X (разликата между тях ще бъде спомената по-надолу, когато се представят *локални* и *нелокални* модули). Избраната операционна система бе Windows, предвид че само такава машина бе налична, а така също че авторът на дипломната работа познава по-добре Windows операционната система.

За написването на C++ кода бе използвана безплатната версия на Microsoft Visual Studio 2010, Express Edition, която обаче практически се използва само като текстов редактор и компилатор, понеже бе преценено, че по-голямата интеграция на тази среда за програмиране ще коства ненужни усилия.

### **3.1. Платформа за създаване на модули за робота Nao**

Платформата на робота Nao позволява лесното създаване на нови модули. Те се делят на два вида – локални и нелокални (отдалечени). Разликата между двата вида модули е, че локалните работят на робота, докато нелокалните вървят на отделна машина, която е свързана с робота по някакъв начин, най-често по мрежата.

Един модул може да осъществява една и съща функционалност независимо дали е локален или нелокален. Всъщност дори е възможно и препоръчително един модул да бъде написан така, че с превключване на един конфигурационен флаг да се компилира като локален или нелокален модул, което пък е наистина лесно постижимо чрез системата за компилация<sup>[16]</sup>. Това се дължи на системата от брокери, които извършват цялата работа свързана с комуникацията между различните модули и операционната система на робота. Това позволява локалните и нелокалните модули да имат еднакъв интерфейс<sup>[17]</sup>.

Работейки на самия робот локалните модули работят по-бързо от нелокалните по отношение на тяхното извикване и достъп до данните на робота, а така също и някои функционалности са ограничени до локални модули<sup>[17]</sup>. От друга страна, локалните модули работят на процесора на робота и използват неговата памет, които не са особено бърз и големи съответно. Нелокалните модули от своя страна могат да работят на произволна друга машина, възможно далеч по-мощна от хардуера на робота. Това според мен създава възможността нелокален модул да работи практически по-бързо от еквивалентния му локален модул, ако самият той извършва тежки пресмятания във вътрешната си логика.

Различните модули, налични на робота в даден момент, локални и нелокални, се достъпват посредством посредници (*proxy*). Повечето предоставени от производителя модули разполагат със собствени посредници,

които осигуряват малко по-бърз достъп до техните методи. Всички останали модули използват посредници от общ вид (всъщност с всички модули може да се работи чрез посредници от общ вид). Посредниците от общ вид позволяват извикването на методи с произволен прототип и резултатна стойност посредством шаблонни функции и подадено като стринг име на метода.

```
boost::shared_ptr<AL::ALBroker> broker =
AL::ALBroker::createBroker("MyBroker", "", 0, robotIP, port);

boost::shared_ptr<AL::ALProxy> testProxy =
boost::shared_ptr<AL::ALProxy>(new AL::ALProxy(broker, "vcoa"));

unsigned int result;
result = testProxy->call<unsigned int>("moveTo", x, y, theta);
```

Самото създаване на нов модул е изключително лесно – всичко, което е необходимо да се направи, е да се създаде клас за новия модул, който да наследява класа `AL::ALModule` и да имплементира два виртуални метода.

```
class vcoa : public AL::ALModule
{
public:
    vcoa(boost::shared_ptr<AL::ALBroker> broker, const std::string &name);
    virtual ~vcoa();
    virtual void init();
};
```

Наследяването на класа от `AL::ALModule` дава всички необходими методи, за да може новият модул да бъде интегриран в системата на робота и да може да бъде извикан от други модули и потребителски приложения.

В конструктора на класа, посредством извикване на функции и макроси, се декларират какви публични методи предлага новият клас – техните имена, параметри и резултатни стойности. Накрая всичко декларирано се свързва с конкретен метод на класа. За да се извика по-късно даден метод, на посредника се подава като стринг декларираното име на метода.

```
functionName("moveTo", getName(), "Move the robot to an (x, y, theta)
position relative to the current position");
addParam("x", "The x goal coordinate, relative to the current position");
addParam("y", "The y goal coordinate, relative to the current position");
addParam("theta", "The theta goal coordinate, relative to the current
position");
setReturn("status", "End status of the command");
BIND_METHOD(vcoa::moveTo);
```

Методът `init` се извиква винаги веднага след конструирането на обект от съответния клас. В него и в деструктора на класа няма задължителни или служебни неща за включване и там създателят на класа може да сложи своята логика за инициализация и дейнициализация.

### **3.2. Модул за памет и събития**

Сред модулите на робота, които са използвани в настоящата работа, е този за памет и отчитане на събития. Той се нарича `AL::ALMemory` и по-главната му задача е да съдържа в себе си информацията за текущото състояние на сензорите и моторите на робота. Освен това обаче той служи и като център за записване и предаване на информация между различните модули, както и за регистрирането на настъпили в системата събития.

Интересно е да се отбележи, че в `AL::ALMemory` събитията представляват обновяване на дадено парче данни (дори и новата стойност да е същата като старата). Това практически означава, че събитията се инициират в момента, в който някой модул промени съответното поле от данни в паметта.

Модулът за откриване и заобикаляне на препятствия не складира информация в този модул и не създава собствени събития, но активно използва информацията и обработва събитията създадени от модула Визуален компас, който ще бъде представен по-долу.

### **3.3. Модул за движение**

Друг модул на робота, който се използва, е този за движение. Макар да не е използван директно стандартният модул за движение `AL::ALMotion`, той вътрешно се извиква от друг използван от мен модул – `AL::ALVisualCompass`.

Модулът за движение предоставя функционалност за движение на робота в равнината на пода. Основният интерфейс на модула представлява команда за движение с три координати,  $x$ ,  $y$  и  $\theta$ , относителни спрямо текущата позиция на робота в момента на получаване на командата. Значението на  $x$  и  $y$  координатите е позиция в двумерна декартова координатна система в равнината на пода, с начало текущата позиция на робота,  $x$ -ос сочеща право напред и  $y$ -ос сочеща точно наляво, като мерната единица е един метър. Значението на  $\theta$  координатата е ориентация на робота, т.е. посока, в която той гледа.

Ориентацията  $\theta$  е зададена като ъгъл на завъртане спрямо ориентацията на робота в момента на получаване на командата, измерен в радиани в посока обратна на часовниковата стрелка. Така команда роботът да се придвижи до позиция (2, -1, 3) би означавала той да отиде до точка, която се намира два метра напред и един метър вдясно от текущата позиция, и роботът да гледа в посока 3 радиана обратно на часовниковата стрелка спрямо първоначалната си посока.

Употребата на този модул спестява на потребителя изключително сложната задача да имплементира самостоятелно ходене. Тази задача включва в себе си управление на множество мотори в тялото на робота, обработка на данните от датчиците му и поддържане на баланса при ходене. Всичко това става автоматично при употребата на този модул.

Един от недостатъците на модула за движение е, че той разчита само на одометрични данни за изчисляване на своята позиция и траектория. Това води до грешки – отклонения от желаната траектория или недостигане на целта. Това се дължи на множество фактори като присъщата неточност при двукракото движение, неточности на датчиците, износване на ставите и мякост на настилката.

Като резултат обикновено роботите се отклоняват в някоя посока от желаната траектория. И при двата ми налични робота това водеше до отклонения вляво при желано движение право напред – роботите описват дъга, когато от тях се иска права. Друг проблем е, че при ходене върху мека повърхност, като мокет например, роботът не успява да задържи добър баланс и се разклатушка още при първите си крачки. Това клатушкане се натрупва и роботът дори може да падне след едва няколко крачки.

### **3.4. Модул Визуален компас**

Основният вграден модул, който е използван в дипломната работа, е AL::ALVisualCompass. Идеята на този модул е, използвайки визуална информация от камерата на робота, да изчисли отклонението на робота от траекторията му при движение право напред. Като следствие от това модулът предоставя и възможност за по-точно придвижване, коригирайки отклонението на робота. Последното става през интерфейс, аналогичен на този на стандартния модул за движение.

Визуалният компас обаче предоставя още неща, от които модулът за откриване и заобикаляне на препятствия се възползва. Това са откриване на ключови точки в изображения, съпоставяне на ключовите точки между две изображения и отсяване на грешните съответствия. Всичко това ще бъде описано в повече подробности при представянето на метода на работа на модула за откриване на препятствия в глава 5, като ще бъде изрично упоменато, че тази работа се извършва от визуалния компас, разработен от Aldebaran.

Накрая трябва да посочим, че едно от нещата, които прави визуалният компас, е да създава събития. Те се отбелязват в модула AL::ALMemory, който извиква обратно функции на регистрираните за събитията абонати, какъвто е модулът за откриване на препятствия. По този начин последният получава възможност да обработи събитията, създадени от визуалния компас, използвайки информацията в тях за собствените си цели.

### **3.5. Модул сонар**

Последният вграден модул на робота, който се използва в дипломната работа, е модулът за сонар, AL::ALSonar. Той отговаря за работата на сонара и също създава събития, които могат да бъдат обработени. Тези събитията са за засечен обект в близост до робота. Неговата употреба се свежда до това да служи като обезопасително средство - при движението на робота чрез модула за откриване и заобикаляне на препятствия, сонарът му постоянно ще сканира близкото пространство около робота и при засичане на препятствие в опасна близост до робота, ще сигнализира за това. Съответно роботът ще спре движението си, предотвратявайки евентуален сблъсък.

Ако модулът за откриване и заобикаляне на препятствия работи акуратно, сонарът не би трябвало да отчита препятствия, но този допълнителен обезопасителен способ не е излишен.

# Глава 4. Анализ на задачата

В тази глава ще направим анализ на проблемите, с които трябва да се справи дипломната работа. Ще започнем с кратък концептуален модел на цялостната стратегия за постигане на желания резултат, включително какъв потребителски интерфейс искаме да притежава модулът. След това ще разгледаме в повече детайли двете основни фази на работа на модула – откриване и заобикаляне на препятствия.

## 4.1. Концептуален модел

Към задачата за откриване и заобикаляне на препятствия може да се подхodi по много начини. Избраният тук подхod се опитва да разгледа едно по-просто решение на проблема, което го прави по-лесно реализуемо и изискващо по-малко изчислителни ресурси. Последното има значение, защото роботът Naо не разполага с голяма изчислителна мощ и дори тя да е достатъчна за справяне със задачата за успешно придвижване, много вероятно е едновременно с това на робота да работят и други процеси, изискващи изчислителна мощ. Пример за такъв процес би бил планировчикът на движенията на робота, който се спомена в глава 1.3.

Самата работа на модула може да се раздели на две части – откриване на препятствия във визуалното поле на робота и заобикалянето им, ако е необходимо. Първото ще се осъществява, докато роботът се придвижва към целта си. При откриването на наличните препятствия ще се прави проверка дали те са разположени на маршрута на робота и в случай, че не са, движението ще продължи нормално. В случай че има блокиращи пътища препятствия, то роботът ще спре и ще се изчисли междинна позиция, към която роботът ще се отправи. Тази междинна позиция е такава, от която очакваме роботът да може да продължи към първоначалната си цел невъзпрепятствано. В случай, че такава позиция не бъде открита, роботът просто ще върне резултат, че не е успял да намери свободен път към целта си.

### 4.1.1. Опростяване на задачата

Трябва изрично да се спомене, че първоначално формулираната задача ще бъде опростена с три основни предположения. Тези презумции улесняват решаваната задача, елиминирайки особености в изображенията, които трябва да бъдат обработвани. Второто предположение ще бъде отново споменато при описание на алгоритъма за откриване на препятствия.

Първото предположение е, че повърхността, по която се движи роботът, няма шарки, които биха били разпознати като особености от алгоритъма за разпознаване на изображения. Това не прави представеното решение напълно непрактично, понеже подове с тази характеристика не са рядкост – едноцветно боядисани подове или чист бетон се срещат сред работните помещения на хората, а ламинатът е популярна подова настилка за дома или офиса.

Второто предположение е, че горната камера, тази която ще бъде използвана, се намира в най-горната точка на робота. Това предположение е свързано с възможността на робота да минава под обекти. Горната камера е разположена в почти най-горната точка на главата на робота, но все пак не е най-отгоре. Това предположение, в почти всички случаи, не би трябвало да създава усложнения, понеже едва ли ще се налага на робота да минава под толкова ниски обекти.

Накрая, най-важното предположение е, че средата около робота ще бъде статична. Това е често срещана конвенция при разработките в роботиката, понеже работата в динамична среда е в пъти по-сложна, което неимоверно вдига трудността на решаваната задача.

#### **4.1.2. Желан потребителски интерфейс**

Потребителският интерфейс, който модулът за откриване и заобикаляне на препятствия искаме да има, е аналогичен по входни параметри на този на стандартния модул за движение и визуалния компас (по-подробно описан в точка 3.3) – задаване на релативна позиция ( $x, y, \theta$ ) към текущата позиция чрез  $x$  и  $y$  координата и ориентация  $\theta$ . Този интерфейс изглежда удачен за всички модули свързани с движение.

Това, което е различно, е връщаната стойност. Модулът за откриване и заобикаляне на препятствия ще дава за резултат код, който представя статуса на изпълнение на командата. Възможните стойности са както следва:

- 0: Успех.
- 1: Неуспех. Причината може да е неуспех на някоя от подкомандите (най-вече тези за движение), засичане на неочекван обект от сонара или друга грешка от общ характер.
- 2: Неуспех поради невъзможност да се открие свободен маршрут до целта.
- 3: Движението е било прекъснато чрез нова команда за движение към модула.

Разширеният списък от резултатни стойности предоставя повече информация на потребителя на модула за причината за неуспех на командата му.

#### **4.2. Откриване на препятствия чрез визуалния компас**

За откриване на препятствия ще бъде използван модула визуален компас. Неговите данни ще бъдат обработени, използвайки паралакс метод и по-точно структура от движение, за да се определят разстояние и ъгъл спрямо наблюдаваните обекти. Това ще позволи да се разберат местоположението на обектите и дали те се намират на пътя на робота.

#### 4.2.1. Работа и резултати на визуалния компас

Описаните в тази точка действия се изпълняват вътрешно от модула визуален компас. Те не са реализация на автора и авторството им принадлежи изцяло на Aldebaran Robotics и авторите на използваните от тях библиотеки. Въпреки това, с цел пълнота на описания алгоритъм, ще бъде преразкан начина им на работа<sup>[12]</sup> и ще бъде описан формата на изходните резултати, които са входните данни на следващата стъпка от алгоритъма за откриване на препятствия.

Първото нещо, което прави визуалният компас, е да заснеме едно изображение, което по-нататък ще наричаме *референтно*. Обикновено това става в началото на движение, зададено на компаса, но може да става и по време на ходене, в който случай референтното изображение се обновява.

В произволен по-късен момент се заснима второ изображение със същите настройки на камерата, каквито са използвани при правенето на референтното изображение. Това ново изображение ще наричаме *текущо*. По време на движението на робота, многократно се взима ново текущо изображение и всички следващи стъпки се изпълняват отново за него, което позволява продължителна корекция на траекторията на ходенето. Така във всеки един момент, за последващите стъпки разполагаме с две изображения, референтно и текущо, които са заснети в различни моменти.

Следващата стъпка е да се извлекат особени (или ключови) точки от двете изображения (за референтното изображение това се прави само веднъж). Алгоритъмът за тяхното извлечане е FAST (*Features from Accelerated Segment Test*, на български Характеристики от ускорен сегментен тест), който засича краища в изображение<sup>[18]</sup>. Използвана е реализацията му в библиотеката OpenCV<sup>[19]</sup>.

Данните за ключовите точки, от които се интересуваме, са:

- Координати  $x$  и  $y$  на центъра на особеността в изображението в пиксели.
- Размер на особеността в пиксели.

След като ключовите точки в двете изображения са извлечени, се прави тяхното съпоставяне едни към други с цел проследяването на един и същ обект в двете снимки. Това става посредством FLANN k-размерно дърво с имплементация от библиотеката OpenCV<sup>[20]</sup>. Резултатът е двойки съпоставени ключови точки, по една от референтното и текущото изображения.

За да се отселят неправилните съпоставления се използва алгоритъм Консенсус със случаен семплиране (RANSAC, *Random sample consensus*)<sup>[21]</sup>. Чрез него се отхвърлят двойките, чието отклонение (от едното към другото изображение) е твърде неконсистентно с общото отклонение на двойките ключови точки. Отстранените двойки се смятат за грешни съпоставки и не се взимат предвид.

Идеята тук е, че всяка двойка представлява един обект, коректно открит и на двете изображения. Изместването на всеки обект се взима предвид и се изчислява общото отместване в заснетите изображения. Това отместване позволява да се коригира траекторията на движение на робота.

По време на работата си, визуалният компас записва резултатите си в паметта на робота и отбелязва събитието, че го е сторил. Това става периодично с различни текущи изображения.

Данните, които визуалният компас предоставя, са:

1. Ключовите точки в референтното и текущото изображения.
2. Двойките съпоставени ключови точки между двете изображения.
3. Кои от двойките ключови точки са консистентни с общото изменение в изображенията. Това ние интерпретираме като правилно съпоставяне.
4. Позицията на робота в момента на заснимане на двете изображения, изчислена чрез одометрия.

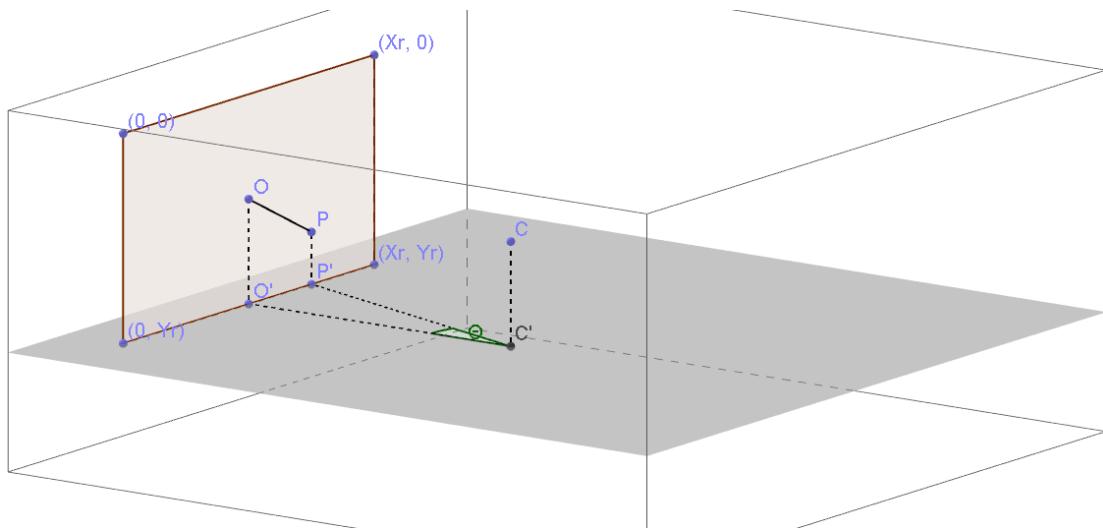
Тук приключва работата на визуалния компас и последващата обработка на резултатите се извършва от модула за откриване и заобикаляне на препятствия.

#### 4.2.2. Определяне на разстояние до обект

За да се проследят обекти между референтното и текущото изображения, се използват съпоставените от визуалния компас двойки ключови точки в двете изображения. Те репрезентират обект, който е заснет и в двете позиции на робота.

Първата стъпка на алгоритъма е да елиминира особеностите, които се намират на ниво над камерата. Тук използваме едно от споменатите по-рано в точка 4.1.1. предположения, че камерата се намира в най-горната точка на робота. И понеже роботът се движи само напред и постоянно приближава обектите в зрителното си поле, то тези, които се намират над камерата, и по предположение над робота, ще намалят у-координатата на пиксела, център на особеността си, между референтното и текущото изображения.

След това филтриране на особените точки, същинската работа на паралакс метода започва.



Фигура 1: Ъгъл в равнината към обект

Предвид, че интерфейсът на робота работи с координати в равнината на повърхността, по която той ходи, реално от интерес за нас са местоположенията на препятствията върху тази равнина, т.е. техните проекции върху равнината. Затова основната характеристика на използванятия в дипломната работа паралакс метод за определяне на разстояние до обект е, че не се търси дистанцията между самия обект и камерата, а между техните проекции върху равнината, по която роботът се придвижва.

На Фигура 1 точката  $C$  представлява позицията на камерата,  $O$  е центърът на изображението,  $P$  е пикселът, център на особеност, а точките  $C'$ ,  $O'$ ,  $P'$  са проекциите на съответните точки в равнината на пода. Понеже камерата на робота гледа напред, то ъгълът  $O'C'P'$  представлява ъгъла, на който роботът трябва да се завърти, за да гледа право към особеността. За да го намерим, използваме зрителното поле на робота (*field of view*), което е известно – 60,97 градуса (тези градуси биват конвертирани в радиани). Търсим ъгъла  $\theta$  като част от цялото зрително поле, като тази част е равна на частта от ширината на изображението, която представлява разстоянието от центъра на изображението до центъра на особеността.

$$\theta = \frac{\theta_v \left( \frac{x_r}{2} - x \right)}{x_r}$$

В тази формула  $\theta_v$  е ширината на зрителното поле,  $x_r$  е ширината на резолюцията на взетото изображение, а  $x$  е ширината в изображението на пиксела, център на особеността.

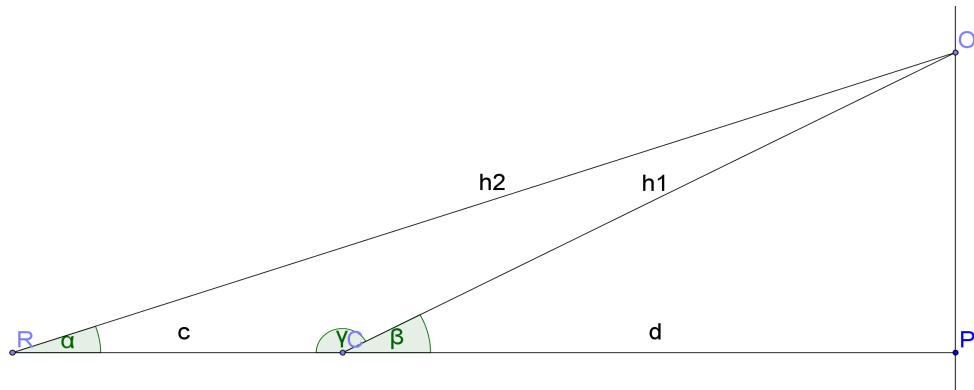
Важно е и да се отбележи, че така полученият ъгъл е в интервала  $\left( -\frac{\theta_v}{2}, \frac{\theta_v}{2} \right]$ , като отрицателните стойности означават, че обектът е вдясно от правата на посоката, в която гледа роботът, а положителните – че обектът е вляво.

След като сме изчислили ъгъла към един обект, от референтното и от текущото изображения, а така също и изминатото разстояние (като разстояние между абсолютните координати на двете позиции), изчисляваме разстоянието до обекта.

Важно е да отбележим още веднъж, че от тук нататък работим само в равнината на пода, поради което логиката ще бъде базирана върху двумерни обекти.

Приемаме, че роботът се движи по права, което, разбира се, практически е невъзможно. Но движението може да бъде достатъчно близко до права, с помощта на визуалния компас в подходящи условия.

Поради шум в движението на робота, напълно е възможно даден обект да попадне от двете страни на тази права при заснемането му от двете позиции, или един или и двета от ъглите да бъдат нулеви. Този случай ще бъде разгледан по-късно. Засега приемаме, че наблюдаваният обект е от една и съща страна на правата и при двете изображения.



Фигура 2: Намиране на разстояние до обект

На Фигура 2 точките  $R$  и  $C$  представляват позициите на робота в моментите на взимане на референтното и текущото изображения,  $O$  е позицията на наблюдавания обект, а  $P$  е проекцията му върху правата на движение. Отсечката  $c$  е изминатото от робота разстояние от момента на вземане на референтното изображение, до момента на вземане на текущото изображение. Това разстояние ни е известно. Търсим разстоянията  $h_2$  и  $h_1$  до обекта от двете съответни позиции, и  $d$  - до проекцията на обекта. Ъглите  $\alpha$  и  $\beta$  са намерените в предходната стъпка ъгли, като отношението помежду им е  $\alpha < \beta$ . Ъгълът  $\gamma$  е допълнителният ъгъл на  $\beta$  и затова  $\gamma = \pi - \beta$ .

От това, че триъгълниците  $RPO$  и  $CPO$  са правоъгълни с прав ъгъл при върха  $P$ , следва, че:

$$\cos \beta = \frac{d}{h_1} \quad \text{и} \quad \cos \alpha = \frac{x+d}{h_2}$$

От тук извеждаме:

$$h_1 = \frac{d}{\cos \beta} \quad \text{и} \quad h_2 = \frac{x+d}{\cos \alpha}$$

От тук, използвайки синусовата теорема за триъгълника  $RCO$  имаме:

$$\frac{h_1}{\sin \alpha} = \frac{h_2}{\gamma} = \frac{h_2}{\sin(\pi - \beta)} = \frac{h_2}{\sin \beta}$$

$$\frac{d}{\sin \alpha \cos \beta} = \frac{x+d}{\cos \alpha \sin \beta}$$

$$d \cos \alpha \sin \beta = (x+d) \sin \alpha \cos \beta$$

$$d (\cos \alpha \sin \beta - \sin \alpha \cos \beta) = x \sin \alpha \cos \beta$$

$$d = \frac{x \sin \alpha \cos \beta}{\cos \alpha \sin \beta - \sin \alpha \cos \beta}$$

Важно е да отбележим, че така полученият резултат е винаги положителен. Това е логично, предвид че  $d$  представлява разстояние, но нека го покажем. Започваме от едно предположение и нататък извеждаме:

$$0 < \alpha < \beta < \frac{\pi}{2}$$

$$0 < \sin \alpha < \sin \beta < 1 \quad \text{и} \quad 1 > \cos \alpha > \cos \beta > 0$$

$$1 > \cos \alpha \sin \beta > \cos \beta \sin \alpha > 0$$

Показахме, че числителят и знаменателят са положителни. Изминатото разстояние  $x$  също е положително, следователно и самото разстояние до проекцията на обекта  $d$  е положително.

Сега, когато имаме разстоянието  $d$ , чрез изведените по-горе формули можем да намерим и разстоянията  $h_1$  и  $h_2$ .

Нека се върнем малко назад и разгледаме случая, в който ъглите към обекта са нулеви. Към този случай ще сведем и случаите, в които ъгълът към обекта е нулев само в едно от референтното и текущото изображения, както и този случай, в който двата ъгъла са с различен знак. За да се получи някой от тези случаи предполагаме, че двата ъгъла са достатъчно малки, за да можем да ги приемем за нулеви.

Когато ъглите към обекта са нулеви, не можем да използваме паралакс метода, защото той разчита на различни и ненулеви ъгли. В този случай разглеждаме феномена перспектива. При него един и същ обект изглежда по-малък, когато се наблюдава от по-далеч, и по-голям - от по-близо. Ще използваме релацията на перспективата между наблюдавания размер на обекта  $s$ , разстоянието до него  $d$  и реалния му размер  $a^{[22]}$ .

$$s = \frac{a}{d} \quad \text{и} \quad a = s d$$

В нашия случай наблюдаваме един и същи обект от две различни разстояния  $d$  и  $(x + d)$  и търсим тези разстояния, съответно по-близо и по-далеч. Размерите на обекта в двете изображенията са съответно  $s_1$  и  $s_2$  и релацията между тях е  $s_1 > s_2$ .

$$d s_1 = a = (x + d) s_2$$

$$d (s_1 - s_2) = x s_2$$

$$d = x \frac{s_2}{s_1 - s_2}$$

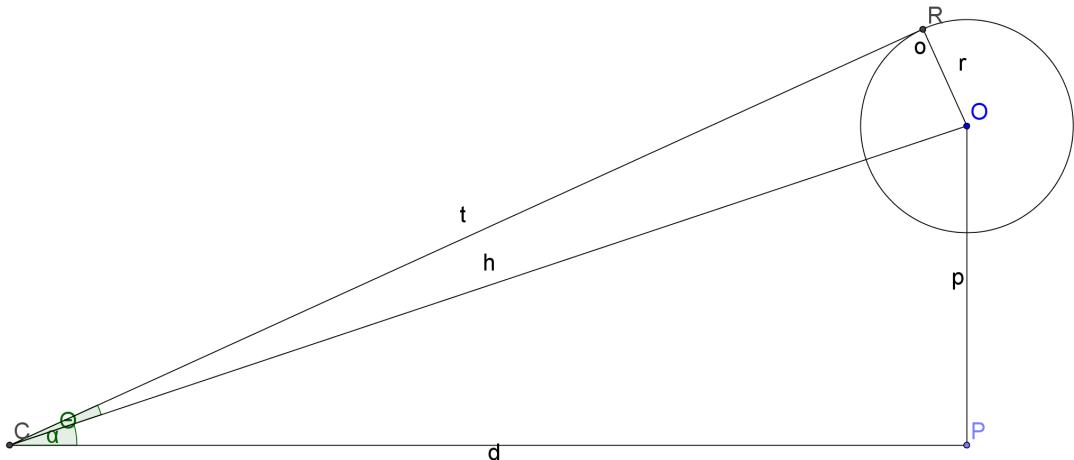
Така намираме разстоянието до обекта, което в случая е равно на разстоянието до проекцията на обекта, понеже ъгълът към него е нулев.

След използването на описания метод, намерихме ъгъла, под който наблюдаваме обект, спрямо правата, по която се движи роботът. Чрез паралакс метод, или в някои случаи чрез перспектива, намерихме и разстоянието до наблюдавания обект и разстоянието до проекцията му върху правата на движение.

### 4.3. Заобикаляне на препятствия

#### 4.3.1. Определяне на блокиращи препятствия

Първата стъпка от алгоритъма за заобикаляне на препятствия е да проверим дали въобще има препятствия, които трябва да заобиколим. За целта още докато откриваме препятствията и изчисляваме разстоянията до тях, пресмятаме и разстоянието между всяко препятствие и правата на движение на робота.



Фигура 3: Разстояние на обект до правата на движение

На Фигура 3 точката  $C$  е текущата позиция на робота,  $O$  е центъра на наблюдаван обект,  $P$  е проекцията му върху правата на движение. Намираме разстоянието  $p$  между центъра на обекта и неговата проекция чрез ъгъла на наблюдаване на обекта  $\alpha$  и разстоянието до него  $h$  по формулата:

$$p = h \sin \alpha$$

Аналогично намираме радиуса  $r$  на обекта по разстоянието до него  $h$  и ъгъла  $\theta$  между допирателната към обекта и отсечката  $CO$ . Този ъгъл е половината от ъгъла от зрителното поле на робота, покрит от обекта. Така имаме формулата:

$$r = h \sin \theta$$

Така разстоянието между обекта и правата на движение е просто  $p - r$ . В случай, че  $p < r$ , приемаме, че разстоянието е нула.

За да разберем дали някое препятствие се намира на пътя на робота, просто трябва да сравним разстоянието от обекта до правата на движение с размера на робота, включвайки и някаква безопасна дистанция, която ще осигури успешното преминаване на робота покрай препятствието и донякъде ще компенсира за шума в данните и някои неточности в положението на точки поради предположения, които сме направили. Дължината на робота Nao с изправени напред ръце е 31,1 см, а широчината му – 27,5 см<sup>[14]</sup>. Така ако представим робота като кръг с диаметър 40 см, предполагаме, че това би било достатъчна дистанция, за да осигури успешното преминаване на робота покрай околните обекти. Понеже кръгът е центриран в точката център на робота, то за сравнението вземаме предвид неговия радиус  $r_r$ , т.е. 20 см. Така ако  $r_r + r < p$ , то роботът, следвайки правата си на движение, би могъл да мине безпроблемно покрай препятствието, понеже то е достатъчно встрани.

Тук си струва да отбележим, че можем да игнорираме препятствия, които лежат на правата на движение на робота, но са твърде далеч, т.е. след целта, понеже те няма да попречат тя да бъде достигната.

Ако няма препятствие, което да пречи на движението на робота по права линия към целта му, то нищо повече не се прави. Ако обаче има препятствие, което лежи на пътя на робота и по този начин го блокира, то се изпраща команда той да спре своето движение и започва процес на изгответяне на маршрут за заобикаляне на препятствието.

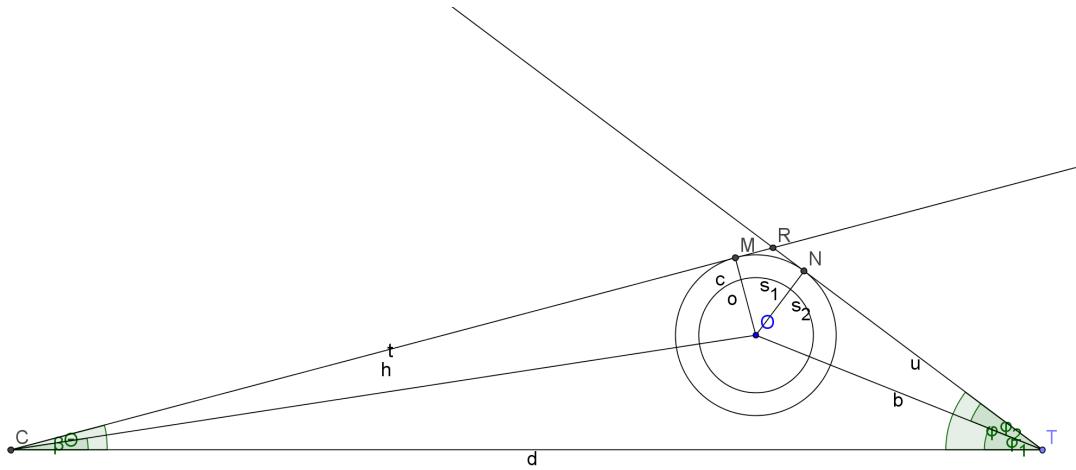
Накрая, възможно е някое препятствие да заема частично или изцяло позицията на целта и по този начин да я блокира. В този случай няма какво да се направи и роботът връща резултат, че няма път до крайната цел, понеже тя не може да бъде достигната.

### 4.3.2. Определяне на междинна позиция

За да заобиколи роботът препятствията, които блокират пътя му, определяме междинна позиция, към която той да се отправи и веднъж достигне ли я, да се насочи отново към първоначалната си цел. Тази междинна позиция е встрани от правата на движение, което ще позволи блокиращото препятствие да бъде заобиколено и по възможност е такава, от която очакваме роботът да може да достигне първоначалната си цел без да е необходимо отново да променя маршрута си.

На Фигура 4 точката  $C$  е текущата позиция,  $T$  е позицията на целта, а  $O$  е центъра на обект. Търсим точката  $R$ , която представлява междинна позиция. Придвижвайки се до нея, роботът ще заобиколи препятствието. За да може роботът успешно да заобиколи препятствието, разглеждаме не кръга на обекта  $o$ , а по-голям кръг  $s$ , чийто център е отново точката  $O$ , но радиусът му  $s$  е по-голям – сбора от радиуса на обекта и безопасната дистанция на робота.

$$s = r + r_r$$



Фигура 4: Ъгли на заобикаляне на препятствие

Правите  $t$  и  $u$  представляват тангенти към кръга  $c$  съответно от текущата позиция  $C$  и целевата позиция  $T$ . Желанието ни е роботът да се движи по тези две прави – по  $t$  от текущата позиция  $C$  до междинната позиция  $R$ , откъдето по  $u$  да се придвижи до целта си  $T$ . Затова и междинната позиция  $R$  се явява тяхната пресечна точка. Именно така и ще я определим – като определим ъглите  $\theta$  и  $\varphi$ , които правите  $t$  и  $u$  сключват с правата на движение на робота.

Определяме ъгъла  $\theta$  като към ъгъла към обекта добавяме ъгъла, на който роботът трябва да се завърти, за да заобиколи обекта:

$$\theta = \beta + \arctan\left(\frac{s}{h}\right)$$

За да определим ъгъла  $\varphi$ , първо намираме разстоянието от обекта до крайната цел  $b$ , използвайки косинусовата теорема за триъгълника  $CTO$ :

$$b = \sqrt{h^2 + d^2 - 2hd \cos \beta}$$

Когато вече имаме  $b$ , използваме синусовата теорема за ъгъла  $\varphi_1$  и формулата за синус за ъгъла  $\varphi_2$ . Ъгълът  $\varphi$  е просто сумата на  $\varphi_1$  и  $\varphi_2$ .

$$\frac{\sin \varphi_1}{h} = \frac{\sin \beta}{b} \quad \text{откъдето} \quad \sin \varphi_1 = \frac{h}{b} \sin \beta$$

$$\sin \varphi_2 = \frac{s}{b}$$

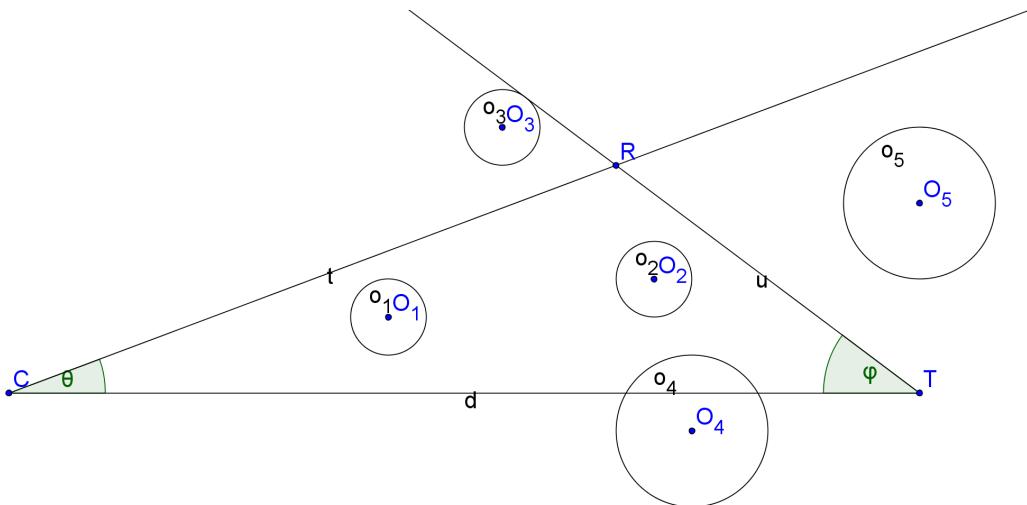
$$\varphi = \varphi_1 + \varphi_2$$

След като имаме формулите, чрез които можем да определим ъглите  $\theta$  и  $\varphi$  за всяко отделно препятствие, остава да се намерят такива ъгли, които да осигурят навигирането на робота между всички тях. За целта за всяко едно препятствие се определя как то трябва да бъде заобиколено. Трябва да се вземе предвид, че някои от тях ще блокират маршрута за заобикаляне на друго

препятствия, а пък други ще са твърде далеч или встани, за да пречат и затова могат да бъдат игнорирани. Алгоритъмът, който се използва, представлява обхождане на всички препятствия и постепенно увеличаване от нула на ъглите  $\theta$  и  $\varphi$  в случай, че разглежданото препятствие го изисква. Така намираме минималните необходими ъгли, които успешно заобикалят всички обекти.

Разглежданите ъгли-решения трябва да са в обсега на зрителното поле и да не налагат връщане назад, т.е. завъртане на повече от прав ъгъл. Така получаваме ограничения, отвъд които не разглеждаме решения и приемаме, че такова не съществува:

$$0 < \theta < \frac{\theta_v}{2} \quad \text{и} \quad 0 < \varphi < \frac{\pi}{2}$$



Фигура 5: Различни позиции на препятствия

На Фигура 5 виждаме различни позиции на препятствия. Намереното решение ги заобикаля успешно по различни начини.

- Обектът  $o_4$  блокира пътя на робота по правата му на движение към целта  $T$ . Поради него се прави първото увеличаване на ъглите  $\theta$  и  $\varphi$ .
- За да се заобиколи обекта  $o_1$ , се налага увеличаване на ъгъла  $\theta$ .
- За да се заобиколи обекта  $o_2$ , се налага увеличаване на ъгъла  $\varphi$ .
- Обектите  $o_3$  и  $o_5$  са твърде встани и далеч съответно и затова не влияят на ъглите  $\theta$  и  $\varphi$ .

Описаната логика се изпълнява за двете страни на правата на движение, лявата и дясната, и по-доброто решение от двете се избира. За по-добро решение приемаме това, чиято сума на ъглите  $\theta$  и  $\varphi$  е възможно най-малка, понеже това ще означава най-малко отклоняване на робота. В случай, че само за една от двете страни има решение, то се използва. Ако за никоя от двете страни не съществува решение, то тогава пътят към целта е изцяло блокиран. В последния случай цялото движение на робота се прекратява и се връща резултат, че път не съществува.

Накрая, ако решение е намерено, то известни са ългите  $\theta$  и  $\varphi$ , както и дължината  $d$  на общата им страна в триъгълника  $CTR$ . Това означава, че той може да бъде определен, с което се намира местоположението на точката  $R$ .

#### **4.4. Изводи**

Накрая на анализа можем да твърдим, че чисто теоретично представеният подход може да бъде решение на задачата за откриване и заобикаляне на препятствия. Въведени са някои опростявания на задачата. Също така сме приели да игнорираме някои несъответствия между описания модел и реалността, които очакваме да са достатъчно малки, за да могат да бъдат компенсирани с използването на по-голям радиус на робота в представянето му в сравнение с реалните му размери. На последното разчитаме и за преодоляване на шума в данните предоставени от робота.

Предстои описаният алгоритъм да бъде имплементиран и тестван, за да се провери приложимостта му в реална среда.

# Глава 5. Проектиране

Желаният потребителски интерфейс, описан в глава 4.1.2., е отправна точка при проектирането на работата на модула за откриване и заобикаляне на препятствия. Затова и той отговаря точно на описаните в тази глава.

Тройката координати  $x$ ,  $y$  и  $\theta$ , която бе описана по-рано в глава 3.3., се нарича *поза* на робота. Допълнително въвеждаме понятията *относителна* и *абсолютна* поза. Теоретично разгледано и двете са относителни, но разликата е в това спрямо какво се разглеждат те. Описаната по-рано поза наричаме *относителна* и тя е с отправна точка текущата позиция на робота. От друга страна, *абсолютна поза* ще наричаме такава поза, която е съотнесена към позицията на робота в момента, в който той е бил включен. И в двата случая се дефинира координатна система в равнината и се задава отправна ориентация. Тези две понятия ще бъдат използвани по-късно.

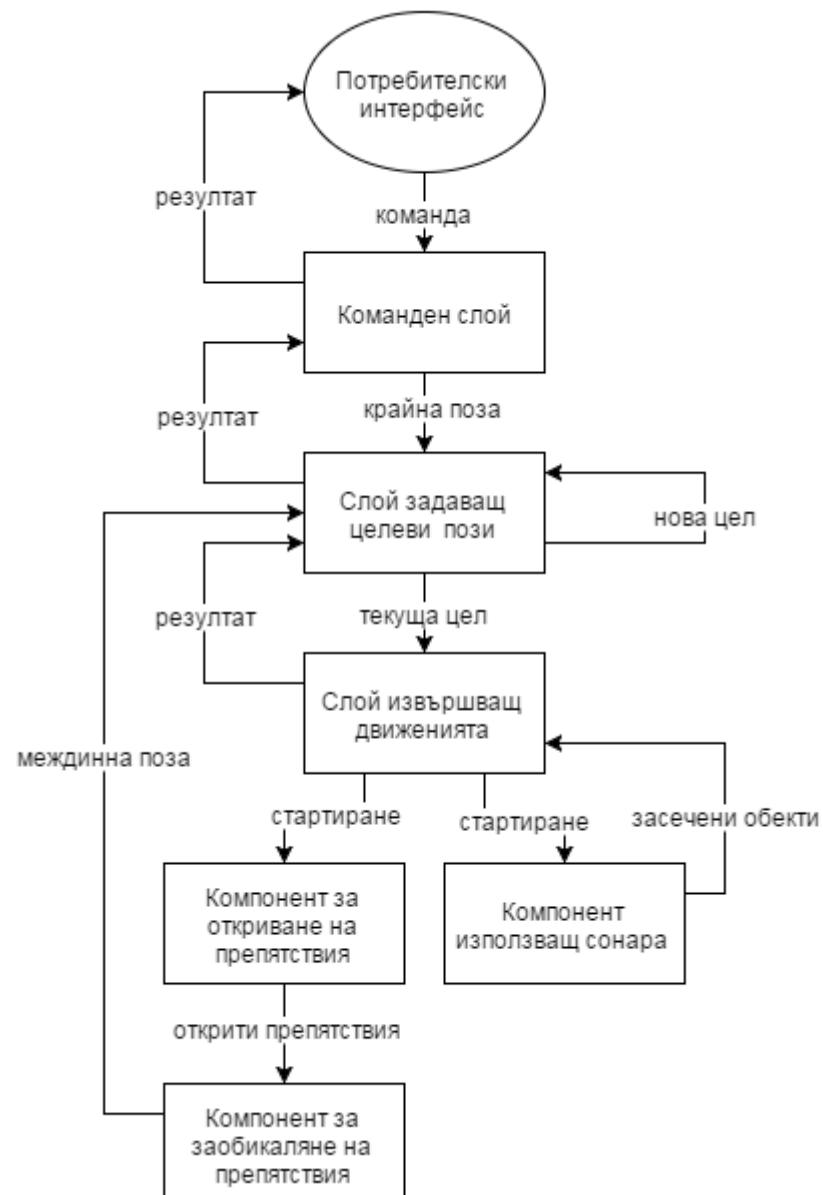
Трябва да се отбележи, че макар в анализа на задачата да се разглеждат точки, в действителност се работи с пози. Причината да се разглеждат точки е, че ориентацията на робота е практически ирелевантна, понеже той може да се завърти на място във всяка една посока. Истински важната част от една поза е местоположението в равнината, което тя задава, т.е. точката зададена от  $x$  и  $y$  координатите на позата. Въпреки това, при проектирането и реализацията на модула, навсякъде се използват пози на робота, а не точки, като понякога ориентацията в позата ще бъде игнорирана.

## 5.1. Обща архитектура

Архитектурата на модула за откриване и заобикаляне на препятствия е проста и е изградена в линейна дълбочина и последователност. Това е постигнато чрез изграждане на итеративен процес на движение – всички движения се извършват едно по едно в цикъл, като всяко следващо започва, едва когато предишното е приключило и е върнало резултат. Именно използвайки върнатия резултат, новото движение получава своята цел.

На Фигура 6 е илюстрирана архитектурата на модула. Най-отгоре стои командният слой, който приема команди от потребителския интерфейс и ги менажира. Под него се намира слоят, задаващ целевите пози – крайната поза или междинна такава, и инициира движенията. Именно в него се намира цикълът, задаващ всяка една цел на робота. Следва слоят, задаващ движенията. Той включва компонентите за откриване и заобикаляне на препятствия и този използващ сонара на робота. След като направи това, този слой извършва движенията по завъртане на робота и движение по права линия. В случай че компонентът за откриване на препятствия засече блокиращи такива, той предава информацията на компонента за заобикаляне на препятствия, който търси междинна позиция, към която роботът да се насочи. В случай че компонентът използващ сонара открие обекти, той връща сигнал, който предизвиква спирането на робота. Всеки един от слоевете връща резултат към слоя над себе си. В някои случаи това би предизвикало ново извикване на по-долния слой.

Крайният резултат от цялата команда бива върнат като резултат през потребителския интерфейс.



Фигура 6: Архитектура на модула

Единственото изключение от линейния в дълбочина модел на модула е, че компонентът за заобикаляне на препятствия, който изчислява междинната позиция, през която роботът да премине, за да заобиколи успешно откритите препятствия, подава резултата си директно на слоя задаващ целевите пози, който не е директно разположен над него. Въпреки това, нагоре се предават само данни и не се инициират никакви процеси, така че това реално не нарушава линейността на архитектурата. Този начин на предаване на данни е направен изцяло за удобство и улесняване на реализацията.

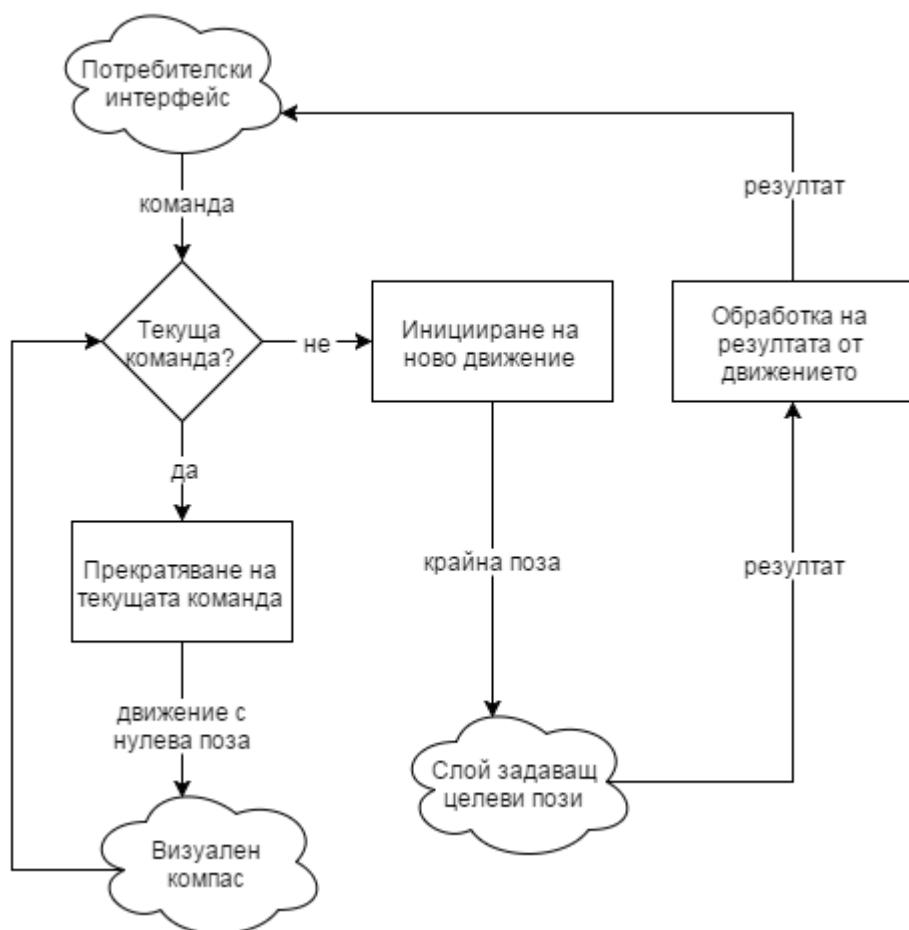
Тук е удачно да се разясни каква разлика се влага в употребата на думите *слой* и *компонент*. *Слоевете* са логически и функционално обособени части от

модула, които са подредени йерархично и комуникацията между тях е ограничена до входни и изходни данни. От друга страна, *компонентите* отново са логически и донякъде функционално обособени части на модула, които обаче работят в контекста на слой – по-свободно използват данните от слоя и могат да се намесват в неговата работа.

Компонентите в разработвания модул са три и всички те работят като част от слоя извършващ движенията. Какво представляват те, каква роля имат и по какъв начин си взаимодействват с този слой, ще бъде обяснено в описанието им по-късно.

## 5.2. Команден слой

На Фигура 7 е представен моделът на работа на командния слой на модула.



Фигура 7: Команден слой

Командният слой е прост и има само две цели. Първата е да обработва резултата на движението, инициирано с команда, дошла от потребителския интерфейс. Това представлява преобразуване на вътрешните резултати на модула към резултатите, които предоставя интерфейса на модула.

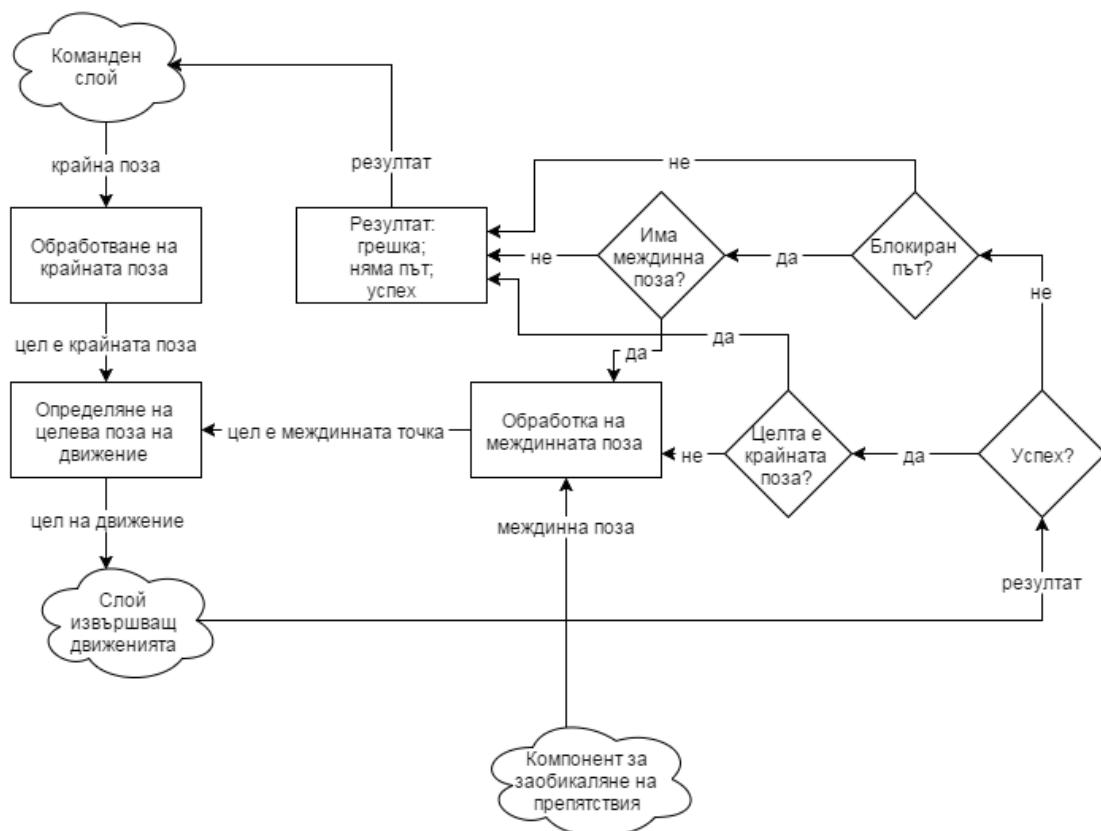
Втората задача е да предостави възможност на потребителя на модула да промени движението на робота. Аналогично на стандартния модул за движение на робота и визуалния компас, ако докато се изпълнява една команда за движение, се получи нова такава, то по-старата бива преустановена и започва да се изпълнява по-новата. Именно с тази задача в модула за откриване и заобикаляне на препятствия е натоварен командния слой.

Този механизъм на смяна на командите има още едно много важно приложение. Практически, за да се накара роботът да спре движението си, на него се подава команда за движение с нулева поза – тройка нулеви координати. Новата команда отменя старата, а понеже нулевата поза не задава никаква промяна в позата на робота, то той няма какво да прави по нея и на практика спира.

Именно по този начин работи и самият модул за откриване и заобикаляне на препятствия, когато той иска да накара робота да спре движението си – подава на визуалния компас нова команда за движение с нулева поза. Този механизъм ще бъде използван и по-нататък в работата на модула.

### 5.3. Слой, задаващ целеви пози

На Фигура 8 е показан моделът на работа на слоя, задаващ целевите пози.



Фигура 8: Слой, задаващ целевите пози

Слойт получава от командния слой крайна поза, до която роботът трябва да се опита да достигне. Обработва я, преобразувайки я от относителна към абсолютна. Това е необходимо, за да може тя да се запази и използва през прекъсвания на движенията.

По-нататък в работата си модулът ще разглежда именно абсолютни пози и ще ги преобразува в релативни, само за да ги задава на визуалния компас.

След като обработи зададената крайна поза, тя се задава като текуща цел и се започва цикъл иницииращ движения към нея. Те се извършват от предназначения за това слой, който връща резултат. В зависимост от този резултат логиката се разклонява.

Ако резултатът е неуспех на движението, то се проверява дали това е заради блокиращ пътя обект. Ако няма такъв, то причината за неуспеха е или спиране на движението от командния слой, или засичане на обект от сонара, или някакъв друг проблем. Във всеки един от тези случаи просто резултатът се връща нагоре към командния слой.

Ако движението е спряло преждевременно поради блокиращ пътя обект, то се проверява дали компонентът за заобикаляне на препятствия е намерил подходяща междинна поза. Ако такава не е била намерена, то се връща резултат, че няма път към целта. В случай, че междинна поза е намерена, то тя се задава като текуща цел и итерацията за това движение приключва. Веднага след това започва нова итерация и ново движение, чиято цел вече е новата междинна поза. Важно уточнение е, че е напълно възможно, докато роботът се придвижва към междинна цел, отново да прецени, че пътят му е блокиран. В този случай той отново има възможност да намери междинна поза, към която да се насочи.

В случай че слойт извършващ движенията върне успех, то се проверява към каква цел се е движил последно роботът. Ако това е крайната поза, то той успешно я е достигнал и към командния слой се връща успех. В противен случай, роботът се е движил към междинна поза, която е била достигната успешно. При това положение за цел се задава отново крайната поза, текущата итерация за движение приключва и започва нова такава.

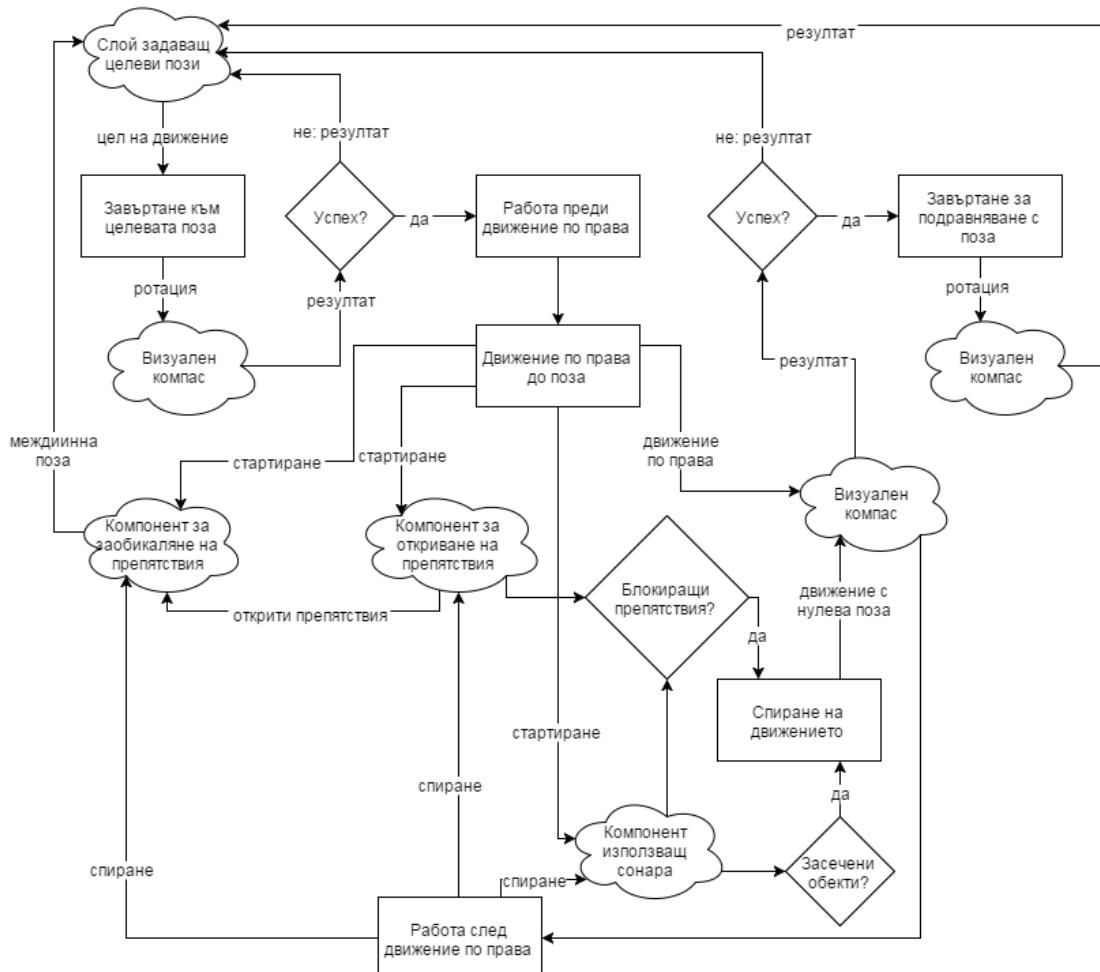
#### **5.4. Слой, извършващ движенията**

На Фигура 9 е показан моделът и последователността на работа на слоя, извършващ движенията.

Макар на схемата да изглежда сложна, последователността на работа на модула всъщност е проста. Когато получи целева поза, той действа аналогично на визуалния компас – разбива движението на три части и ги изпълнява последователно. Първо роботът се завърта към целевата поза, така че до нея да се движи само по права линия. След като направи това, по въпросната права линия роботът се придвижва определено разстояние. Накрая, той се завърта, така че да гледа в желаната от целевата поза посока. Всяко от тези действия се задава за изпълнение на визуалния компас. Ако някое от действията претърпи неуспех, цялата работа се прекъсва и неуспехът се връща към горния слой.

Разликата е, че покрай движението по права линия са закачени допълнителни неща. От гледна точка на собствената логика на този слой, това са само включване преди движението и изключване след него на компонентите

за откриване и заобикаляне на препятствия и компонента, използващ сонара. Тези компоненти реално работят в контекста на слоя и имат възможността да спират движението. Те ще бъдат описани по-подробно в следващите глави.



Фигура 9: Слой, извършващ движенията

Компонентите за откриване и заобикаляне на препятствия реално са едно цяло, но понеже реализират сравнително обособена логика, са разгледани поотделно.

Трите компонента, заедно със слоя задаващ целевите пози, са мястото, където темата на дипломната работа е приложена и имплементирана.

#### 5.4.1. Компонент за откриване на препятствия

Компонентът за откриване на препятствия имплементира логиката за откриване на препятствия, описана в цялата глава 4.2. - това е анализиране на данните, предоставени от визуалния компас, и определяне на разстоянието до заснетите обекти. Допълнително, в този компонент е включена логиката от глава 4.3.1., отнасяща се до определяне дали пътят на робота е блокиран или не.

Моделът на работа на компонента е описан подробно в предишните глави и затова няма да бъде разглеждан тук. Единствено ще посочим, че ако пътят на робота се окаже блокиран от обекти, той изпраща команда за спиране към визуалния компас, т.е. движение с нулева поза. Това ще прекъсне текущата команда към него, зададена от слоя, извършващ движенията. Това ще означава, че слоят ще получи за резултат неуспех и той самият ще върне неуспех нагоре към слоя, даващ целевите пози. Той от своя страна ще изчака компонентът за заобикаляне на препятствия да завърши своя анализ и да подаде резултатите си. Така ще се задвижи по-рано описаната в глава 5.3. логика на слоя, определящ целевите пози.

#### **5.4.2. Компонент за заобикаляне на препятствия**

Компонентът за заобикаляне на препятствия имплементира логиката за заобикаляне на препятствия, описана в глава 4.3.2. - определяне на междинна поза. Моделът на работа на компонента е описан подробно в предишната глава и затова няма да бъде разглеждан тук. Само ще уточним, че този компонент получава намерените обекти от компонента за отриване на препятствия (те практически са едно цяло) и изпраща резултата си, междинната поза или това, че няма такава, на слоя, определящ целевите пози.

#### **5.4.3. Компонент, използващ сонара**

Работата на компонента използващ сонара на робота е много прости и вече няколко пъти бе спомената – ако при движението си по правата сонарът засече някакви обекти, той спира движението, за да предотврати колизия. Това спиране отново става чрез изпращане на движение с нулева поза, както е описано по-рано.

# Глава 6. Реализация и тестване

## 6.1. Реализация на слоевете и компонентите

Слоевете и компонентите са реализирани, следвайки идеите заложени при проектирането. В тази глава ще представим части от имплементацията им и ще посочим някои трудности или по-интересни парчета код.

Модулът за откриване и заобикаляне на препятствия се нарича `vcoa` (*Visual Compass Obstacle Avoidance*, или на български *Заобикаляне на препятствия чрез визуален компас*). Както беше посочено в глава 3.1., той наследява класа `AL::ALModule`.

```
class vcoa : public AL::ALModule
```

Класът `vcoa` предоставя публичен интерфейс за работа с него, както и съдържа частни член данни и методи. Част от тях ще бъдат представени в изложението на тази глава.

Като базов скелет на служебната част от кода на модула е използван код, взет от примерен модул от сайта на Aldebaran.<sup>[16][23]</sup>

### 6.1.1. Потребителски интерфейс

Желаният потребителски интерфейс, представен в глава 4.1.2., е реализиран. Създаването на обект от тип `vcoa` става чрез следния конструктор:

```
vcoa(boost::shared_ptr<AL::ALBroker> broker, const std::string &name);
```

След като е създаден обект от типа `vcoa`, към него могат да се подават команди за движение чрез метода му `moveTo`. Неговите аргументи и резултатни стойности са посочени и обяснени в следния код:

```
enum result_t
{
    moveSuccess,
    moveFailure,
    noPathFound,
    aborted
};

/// <summary>
/// Go to input pose (in robot referential).
/// </summary>
/// <param name="x"> Distance along the X axis in meters.</param>
/// <param name="y"> Distance along the Y axis in meters.</param>
/// <param name="theta"> Rotation around the Z axis in radians [-3.1415 to
3.1415].</param>
/// <returns>The result of the movement. One of the result_t enum
values</returns>
unsigned int moveTo(const float& x, const float& y, const float& theta);
```

За момента са дефинирани само тези четири резултатни стойности, но към тях могат да се добавят още. Например да се специфицира причината за неуспех – дали роботът не е успял да извърши дадена част от движението или сонарът е засякъл опасно близък предмет.

### 6.1.2. Команден слой

Командният слой е поместен в метода `moveTo`, който е началната точка на заявките към модула. Основната му задача е да прекратява старите движения, когато се получат команди за нови такива. Това става лесно, чрез изпращане на движение с нулева поза към визуалния компас. По-трудната задача е да се определи дали в момента има изпълняваща се команда и най-вече всичко това да се синхронизира правилно. Проблемът тук е, че различните команди се изпълняват в отделни нишки, които трябва да се синхронизират по такъв начин, че нишката на новата команда да изчака приключването на работата на нишката на старата команда, преди да започне изпълнение на собственото си движение. Това става с помощта на следните член данни:

```
boost::mutex statusMutex;
boost::condition_variable commandCondvar;

unsigned char command;
```

Логиката за синхронизация е проста – при постъпване на команда за движение се заема мутекса `statusMutex` и се проверява дали вече има изпълняваща се такава, като се проверява стойността на члена `command`. В случай, че тя е различна от `COMMAND_IDLE`, т.е. има изпълняваща се команда, стойността се променя на `COMMAND_ABORT`, подава се движение с нулева поза към визуалния компас и се чака, докато стойността на `command` стане `COMMAND_IDLE`, т.е. вече няма друга изпълняваща се команда. Когато няма друга команда, стойността на `command` се променя на `COMMAND_MOVING`, за да се сигнализира на евентуални нови команди към модула, че в момента се изпълнява движение. Мутексът се освобождава и заявката се препредава на по-долния слой (метода `moveToRelative`). Когато тя приключи, независимо с какъв резултат, мутексът отново се заема, на `command` се дава стойност `COMMAND_IDLE` и се нотифицират всички нишки чакащи на условната променлива `commandCondvar`. Накрая мутексът се освобождава и методът веднага приключва работа. Ако е имало чакащи нишки, то те са чакали да се спре изпълняващата се команда в случая, описан по-горе, с което механизъмът за синхронизация е завършен.

```
unsigned int vcoa::moveTo(const float& x, const float& y, const float& theta)
{
    // Abort any previous move.
    {
        boost::unique_lock<boost::mutex> lock(this->statusMutex);
        while (this->command != COMMAND_IDLE)
        {
            this->command = COMMAND_ABORT;
            this->compassProxy->moveTo(0, 0, 0);      // Abort any ongoing
compass move command.
```

```

        this->commandCondvar.wait(lock);
    }

    this->command = COMMAND_MOVING;
}

this->moveToRelative(x, y, theta);

unsigned int result;
/* Process result */

this->command = COMMAND_IDLE;

this->commandCondvar.notify_all();

return result;
}

```

Втората задача на командния слой е да преобразува вътрешния за модула резултат към стойностите, които се връщат към потребителския интерфейс.

### 6.1.3. Метод за извлечане на текущата поза на робота

Много важно място в работата на модула vcoa заема помощният метод `getCurrentWorldPose`. Той взема текущата абсолютна поза на робота посредством неговите датчици.

```

bool vcoa::getCurrentWorldPose(AL::Math::Pose2D& pose)
{
    AL::ALValue deviation;

    deviation = this->memoryProxy->getData("VisualCompass/Deviation");

    /* Some error checking */

    pose = (AL::ALValue::TFloatArray)deviation[1];

    return true;
}

```

Данните се съхраняват в паметта на робота от полето `VisualCompass/Deviation`. Извличат се посредством метода `getData` на член променливата за посредник на паметта `memoryProxy`.

```
boost::shared_ptr<AL::ALMemoryProxy> memoryProxy;
```

Поради неизвестни причини, тези данни не винаги са налични, което налага проверки за грешки при извлечането им и за консистентност.

Получаването на текущата абсолютна поза на робота позволява преобразуването на релативни пози в абсолютни, както и намирането на разликата между две текущи пози на робота, което се използва за намиране на изминатото разстояние между тях. Накрая, понеже визуалният компас работи с релативни пози, посредством вземането на текущата абсолютна поза, може да се изчисли релативната поза, която ще транслира текущата абсолютна в целевата

абсолютна поза. Така определената релативна поза трябва да се зададе на визуалния компас, така че той да придвижи робота по желания начин.

#### 6.1.4. Слой, задаващ целеви пози

Имплементацията на слоя, задаващ целевите пози, е поместена в метода moveToRelative. Тя е отново проста и следва проектираното.

```
void vcoa::moveToRelative(const float& x, const float& y, const float&
theta)
{
    bool result;

    AL::Math::Pose2D relativeGoalPose(x, y, theta);
    AL::Math::Pose2D zeroPos(0);
    float distance = relativeGoalPose.distance(zeroPos); // The distance
between the current and the goal poses.
    float angle = AL::Math::modulo2PI(relativeGoalPose.getAngle()); // The
angle that the robot should rotate
                                         // in
order to face the goal pose.

    // Get the starting 2D pose.
    if (!this->getCurrentWorldPose(this->worldStartPose))
    {
        this->status = STATUS_FAILURE;
        return;
    }

    // Compute the goal 2D pose.
    float auxAngle = AL::Math::modulo2PI(this->worldStartPose.theta +
angle);
    this->worldGoalPose.x = this->worldStartPose.x + distance *
cos(auxAngle);
    this->worldGoalPose.y = this->worldStartPose.y + distance *
sin(auxAngle);
    this->worldGoalPose.theta = AL::Math::modulo2PI(this-
>worldStartPose.theta + theta);

    this->status = STATUS_GOAL;
    this->sonarDetectedObstacle = false;

    bool exitLoop;
    do
    {
        this->moveToAbsolute();

        boost::unique_lock<boost::mutex> lock(this->statusMutex);
        while (this->status == STATUS_INTERRUPT)
        {
            this->statusCondvar.wait(lock);
        }

        exitLoop = (this->status != STATUS_GOAL) && (this->status !=
STATUS_REROUTE);
    }
    while (!exitLoop);
}
```

Първата работа на метода е да зададе начални стойности на някои променливи и да преобразува релативната крайна поза към абсолютна. Това се

прави, за да може тя по-лесно да се запази между различните итерации на цикъла за движение към цел. Това става, като първо се вземе текущата поза, след което тя се отмества с релативната поза. Така получаваме крайната поза като абсолютна.

Резултатът се записва в член променливата `worldGoalPose`, която по-нататък се използва от останалите методи на класа. Аналогично в член променливата `worldTransitPose` се записва междинната позиция, когато има такава. Именно по този начин компонентът за заобикаляне на препятствия предава изчислената от него поза, когато има такава.

```
AL::Math::Pose2D worldGoalPose;  
AL::Math::Pose2D worldTransitPose;  
  
unsigned char status;
```

След като има крайната цел в абсолютни координати, методът я задава като текуща цел посредством задаване стойност `STATUS_GOAL` на член променливата `status`, след което започва цикълът от движения към абсолютни целеви пози, който прави извиквания към слоя, извършващ движенията (метода `moveToAbsolute`).

Тук важна отново е синхронизацията, защото ако движението бъде прекъснато поради блокиращ пътя обект, възможно е изчисляването на междинна позиция да отнеме повече време. Това се сигнализира чрез стойност `STATUS_INTERRUPT` на променливата `status`, зададена от компонента за откриване на препятствия. Посредством изчакване по условната променлива `statusCondvar` и проверка на стойността на `status` се синхронизира, че работата по търсенето на междинна поза е приключила.

Когато стойността на `status` стане различна, тя се проверява и ако е една от `STATUS_REROUTE` или `STATUS_GOAL`, то се извършва нова итерация на цикъла. Тези две стойности показват към каква поза предстои роботът да се движи търпърва – към междинна поза, понеже пътят му е бил блокиран, или към крайната поза, след като последно е преминал през междинна поза. Ако стойността на `status` е различна, то или движението е претърпяло неуспех поради някаква причина, или крайната поза е била достигната, в който случай имаме успех. И в двете ситуации цикълът от движения приключва, както и целия метод.

Стойността на член променливата `status` се задава от компонента за заобикаляне на препятствия, в случай че има блокиращ пътя обект, или от слоя извършващ движенията, ако движението на робота завърши поради друга причина.

### 6.1.5. Слой, извършващ движенията

Слойт, извършващ движенията, е имплементиран в метода `moveToAbsolute`, който от своя страна извиква три помощни метода, които ще бъдат описани по-късно. Първата стъпка е да се провери към коя цел трябва да се запъти робота, чрез проверка на член променливата `status`. Това става като първо се заема мутекса `statusMutex`, след което се проверява стойността на

status. Ако тя е STATUS\_REROUTE, то цел ще е междинната поза, а на status се дава нова стойност STATUS\_TRANSIT (промяната е необходима поради синхронизационни причини). В противен случай, цел ще е крайната поза. След проверката мутексът statusMutex се освобождава.

```
AL::Math::Pose2D worldTargetPose;
{
    boost::unique_lock<boost::mutex> lock(this->statusMutex);
    if (this->status == STATUS_REROUTE)
    {
        this->status = STATUS_TRANSIT;
    }
    worldTargetPose = (this->status == STATUS_GOAL) ? this->worldGoalPose :
    this->worldTransitPose;
}
```

След като целта е определена, се извършва първата част от движението – ротация към целевата поза, т.е. така че роботът да гледа право към целта си. За да се постигне това се извиква помощния метод compassRotateTowards. Ако ротацията е била неуспешна, цялото движение се прекратява.

```
// Rotate accordingly so that the desired pose is straight ahead.
result = this->compassRotateTowards(worldTargetPose);
if (!result)
{
    boost::unique_lock<boost::mutex> lock(this->statusMutex);
    this->status = STATUS_FAILURE;
    return;
}
```

Ако ротацията е била успешна, то следва движение по права линия, докато се достигне целта. Именно тук е особеността, понеже преди да започне движението, се включват компонентите за откриване и заобикаляне на препятствия и компонента, работещ със сонара. И трите представляват функции за обратна връзка, които се регистрират като абонати на събития на визуалния компас. След като модулът се е абонирал за съответните събития, движението по права линия се извършва от помощния метод compassMoveStraightTo, след което абонаментите се прекратяват веднага и едва тогава се анализира резултатът от движението. Ако той показва неуспех, то цялото движение се прекратява.

```
// Turn on the sonar and subscribe to events only while moving straight.
this->sonarProxy->subscribe("vcoa");
this->memoryProxy->subscribeToMicroEvent("SonarLeftDetected", "vcoa", "", "onSonarDetection");
this->memoryProxy->subscribeToMicroEvent("SonarRightDetected", "vcoa", "", "onSonarDetection");
this->memoryProxy-
>subscribeToMicroEvent("VisualCompass/NewReferenceImageSet", "vcoa", "", "onNewRefImageSet");
this->memoryProxy->subscribeToMicroEvent("VisualCompass/Deviation", "vcoa", "", "onDeviation");

// Move straight ahead to the desired pose.
result = this->compassMoveStraightTo(worldTargetPose);
```

```

// Turn off the sonar and unsubscribe from the events before analyzing the
result of the move.
this->memoryProxy->unsubscribeToMicroEvent("VisualCompass/Deviation",
"vcoa");
this->memoryProxy-
>unsubscribeToMicroEvent("VisualCompass/NewReferenceImageSet", "vcoa");
this->memoryProxy->unsubscribeToMicroEvent("SonarRightDetected", "vcoa");
this->memoryProxy->unsubscribeToMicroEvent("SonarLeftDetected", "vcoa");
this->sonarProxy->unsubscribe("vcoa");

if (!result)
{
    boost::unique_lock<boost::mutex> lock(this->statusMutex);
    // If the move ended for a reason other than an interrupt, treat it as
failure.
    if ((this->status != STATUS_INTERRUPT) && (this->status !=
STATUS_REROUTE) && (this->status != STATUS_NO_PATH))
    {
        this->status = STATUS_FAILURE;
    }
    return;
}

```

Компонентите за откриване и заобикаляне на препятствия са имплементирани като едно цяло, а те самите имат логика при събитията `VisualCompass/NewReferenceImageSet` и `VisualCompass/Deviation`. Компонентът, използващ сонара, се активира при някое от събитията `SonarLeftDetected` или `SonarRightDetected`, като и при двете се изпълнява една и съща логика.

При настъпване на някое от тези събития, логиката на въпросния компонент се изпълнява. Имплементацията на всеки от тези компоненти ще бъде представена поотделно в следващите глави.

Ако и движението по права е било успешно, то роботът е пристигнал в целевата точка. Остава той да се завърти по такъв начин, че ориентацията му да съвпада с тази на целевата поза. Това се извършва от помощния метод `compassRotateAlignWith`. След като ротацията приключи, се прави анализ на резултата и с това цялото движение е свършило и методът излиза.

```

// Rotate accordingly to face the desired end orientation.
result = this->compassRotateAlignWith(worldTargetPose);

boost::lock_guard<boost::mutex> lock(this->statusMutex);
if (!result)
{
    this->status = STATUS_FAILURE;
}
else if (this->status == STATUS_GOAL)
{
    // The goal pose has been reached.
    this->status = STATUS_SUCCESS;
}
else if (this->status == STATUS_TRANSIT)
{
    // The transit pose has been reached. Move to the goal pose.
    this->status = STATUS_GOAL;
}
// else do not change the status, as it is controlled by an interruption.

```

Ако ротацията е била неуспешна, то резултатът от цялото движение е неуспех. Но ако тя е била успешна, то в зависимост от това до коя цел се е придвижила роботът, се задава или че цялата команда е била успешна, или че роботът сега трябва да се насочи към междинната поза.

Действието и на трите помощни метода `compassRotateTowards`, `compassMoveStraightTo` и `compassRotateAlignWith` е аналогично. Първо се проверява дали не е пристигнала нова команда, която да отмени текущата. След което визуалният компас се инструктира да смени референтното си изображение с цел той да работи с по-актуални изображения и най-вече за да може движението по права да разглежда изображения, които са снимани от позиции, лежащи на въпросната права. Взема се и текущата поза. Тази последователност от действия е една и съща и за трите метода.

```
{
    // Check for abort command.
    boost::unique_lock<boost::mutex> lock(this->statusMutex);
    if (sonarDetectedObstacle || (this->command == COMMAND_ABORT))
    {
        return false;
    }
}

bool result = this->compassProxy->setCurrentImageAsReference(); // Before
each compass move set a new reference image.

AL::Math::Pose2D worldCurrentPose;
if (!this->getCurrentWorldPose(worldCurrentPose))
{
    return false;
}
```

Едва тук се появяват различия между трите помощни метода – пресмятата се разликата между текущата и целевата пози и се задава съответната команда за ротация или движение напред на визуалния компас.

В метода `compassRotateTowards` кодът е:

```
float angle = AL::Math::modulo2PI((worldTargetPose -
worldCurrentPose).getAngle() - worldCurrentPose.theta);

result = this->compassProxy->moveTo(0, 0, angle);
```

В метода `compassMoveStraightTo` кодът е:

```
float distance = worldCurrentPose.distance(worldTargetPose);

result = this->compassProxy->moveStraightTo(distance);
```

В метода `compassRotateAlignWith` кодът е:

```
float angle = AL::Math::modulo2PI(worldTargetPose.theta -
worldCurrentPose.theta);

result = this->compassProxy->moveTo(0, 0, angle);
```

И трите метода завършват с връщане на резултата си.

## 6.1.6. Компонент за откриване на препятствия

Първата част от компонента за откриване на препятствия е функцията за обратна връзка `onNewRefImageSet`, която се стартира при настъпването на събитието `VisualCompass/NewReferenceImageSet`. В тази функция, всъщност метод на класа `vcoa`, при смяната на референтното изображение, се запомня текущата поза на робота. Тази поза по-нататък се нарича референтна поза. Достъпът до нея се осъществява само след заемане на мутекса `refPoseMutex`.

```
void vcoa::onNewRefImageSet(std::string /* eventName */, bool isSet,
                           std::string /* subscriberIdentifier */)
{
    if (!isSet)
    {
        return; // Ignore reference image cleared events.
    }

    AL::Math::Pose2D worldCurPose;
    if (this->getCurrentWorldPose(worldCurPose))
    {
        this->refPoseMutex->lock();
        this->worldRefPose = worldCurPose;
        this->refPoseMutex->unlock();
    }
}
```

Компонентите за откриване и заобикаляне на препятствия представляват едно последователно цяло. Първият започва същинската си работа при настъпването на събитието `VisualCompass/Deviation` във функцията за обратна връзка `onDeviation` (отново всъщност метод на класа `vcoa`), а след приключване на своята работа, този метод продължава, започвайки в него своята работа компонентът за заобикаляне на препятствия.

Събитието `VisualCompass/Deviation` се инициира в момента, в който отклонението в движението на робота се пресметне от визуалния компас. Това се случва и след като са направени съпоставките между двойките точки от референтното и текущото изображения (което инициира събитието `VisualCompass/Match`, за което обаче няма абониран метод). Това означава, че данните във `VisualCompass/Match` също са били обновени и могат да бъдат използвани с актуални стойности (припомняме, че събитията представляват сигнал, че съответното парче данни в паметта е било обновено).

Първото нещо, което се прави при извикването на функцията за обратна връзка `onDeviation`, е да се извлече текущата поза. Това се прави веднага и се ползват данните от събитието, а не от помощния метод `getCurrentWorldPose`, за да се избегне евентуална промяна в позата на робота. Взимат се също така и референтната и целевата пози.

```
// Copy the deviation information, as it might change.
AL::ALValue deviation = deviationRef;

AL::Math::Pose2D worldEventPose = (AL::ALValue::TFloatArray)deviation[1];

AL::Math::Pose2D worldRefPose, worldTargetPose;
this->refPoseMutex->lock();
worldRefPose = this->worldRefPose;
this->refPoseMutex->unlock();
```

```

{
    boost::unique_lock<boost::mutex> lock(this->statusMutex);
    worldTargetPose = (this->status == STATUS_GOAL) ? this->worldGoalPose :
    this->worldTransitPose;
}

```

Понеже събитието се случва ненужно често, част от събитията могат да се пропуснат и е по-добре това да се направи, понеже така ще се спестят ненужни изчисления – събитията се появяват едно след друго без роботът практически да е променил съществено позицията си. Това се постига по два начина. Първият е като се използва условно заемане на мутекса `matchMutex`, което става, само ако мутексът е свободен. Разликата с нормалното заемане на мутекса е, че по нормалния начин извикването е блокиращо и ако мутексът е взет в момента, то се изчаква той да се освободи. При условното заемане на мутекса в описания случай извикването няма да блокира, а веднага ще върне, че мутексът в момента не е свободен.

```

bool sectionLocked = true;
AL::ALCriticalSection deviationSection(this->matchMutex,
sectionLocked);
if (sectionLocked)
{
    return;
}

```

Вторият начин за елиминиране на ненужни събития е, като се въведе минимално изискване за промяна в позицията на робота, за да се изпълни въобще логиката на компонента. В случая е използвано минимално разстояние от 5 см.

```

float refDistance = worldEventPose.distance(worldRefPose);
if (refDistance < 0.05)
{
    return; // Skip the event because of too little deviation.
}

```

Записва се още информацията от съпоставянето на двойките особени точки, след което идва същинската работа на метода – намирането на препятствията.

```

// Get the current match information. This is the closest to correct match
information we can get.
AL::ALValue matchInfo;
matchInfo = this->memoryProxy->getData("VisualCompass/Match");

/* Some error checking */

// Go through each keypoint (obstacle) and calculate the distance and angle
to it.
int resolutionId = this->compassProxy->getResolution();
std::vector<Obstacle> obstacles;
findObstacles(refDistance, resolutionId, matchInfo, obstacles);

/* Some error checking */

```

Помощната функция `findObstacles` обхожда всички двойки точки от двете изображения. Първо преценява дали те трябва да бъдат разглеждани – обектите, които на височина са по-нагоре от робота, биват игнорирани съгласно опростяването, представено в глава 4.1.1. В показания код точките имат на индекс 0 координатите си в изображението, от които индекс 1 е  $y$ -координатата.

```

for (int i = matchInfo[3][1].getSize() - 1; i >= 0; --i)
{
    int matchIndex = matchInfo[3][1][i];
    // NOTE: According to the documentation these two should have their
    values switched,
    // but experiments show that they have these values.
    int refIndex = matchInfo[2][matchIndex][1];
    int curIndex = matchInfo[2][matchIndex][0];

    if (matchInfo[0][refIndex][0][1] < matchInfo[1][curIndex][0][1])
    {
        // The keypoint is above the robot camera, so consider it higher
        than the robot and therefore skip it.
        continue;
    }

    /* More match processing */
}

```

Следващата стъпка е да се изчислят ъглите в равнината спрямо обектите в референтното и текущото изображения. Това става с помощта на функцията `calculateHorizontalAngleTowardsFieldOfViewPoint`, на която се подава  $x$ -координатата на точка в изображение и  $x$ -размерът на изображението.

```

const float CAMERA_HORIZONTAL_FOV_ANGLE = (float)(60.97 * AL::Math::PI /
180.0); // In radians.

// Calculates the horizontal (in the ground plane) angle towards a shot by
// the camera point in the robot field of view.
static float calculateHorizontalAngleTowardsFieldOfViewPoint(int pointX, int
resX)
{
    return CAMERA_HORIZONTAL_FOV_ANGLE * (float)(resX / 2 - pointX) /
(float)resX;
}

```

Следва намиране на разстоянията до обекта и до проекцията на обекта на правата на движение. Това става по два различни начина, в зависимост от това какви са намерените ъгли.

```

if (obstacleRefImAngle * obstacle.angle <= 0)
{
    // An angle is zero or the two angles have a different sign.
    // Treat both cases as if both angles are zero,
    // i.e. in both images the object center lies on the straight line the
    robot walks on.
    obstacle.directDistance =
        calculateDistanceWith0Angle(matchInfo[0][refIndex][1], matchInfo[1]
[curIndex][1], displacement);
    obstacle.projectionDistance = obstacle.directDistance;
}
else
{

```

```

obstacle.projectionDistance =
    calculateProjectionDistance(abs(obstacleRefImAngle),
abs(obstacle.angle), displacement);
    obstacle.directDistance = obstacle.projectionDistance /
sin(obstacle.fovCover);
}

```

Помощните функции calculateDistanceWith0Angle и calculateProjectionDistance са дефинирани, както следва:

```

// Calculates the distance to a point on the straight ahead line that the
robot walks on, based on sizes in images.
static float calculateDistanceWith0Angle(float refImSize, float curImSize,
float displacement)
{
    return displacement * refImSize / (curImSize - refImSize);
}

// Calculates the distance to a point projection on the straight ahead line
that the robot walks on.
static float calculateProjectionDistance(float refImAngle, float curImAngle,
float displacement)
{
    return displacement * sin(refImAngle) * cos(curImAngle) /
(cos(refImAngle) * sin(curImAngle) - sin(refImAngle) *
cos(curImAngle));
}

```

Изчисляват се още ъгълът от зрителното поле, което даден обект покрива (*field of view cover*), и радиусът на обекта. Тук е удачно част от обектите да се филтрират, използвайки минимален радиус. По този начин се елиминират някои много малки обекти, които вероятно са или шум, или незначителни шарки по пода. За минимален радиус е използвана граница от 1 см. Накрая обектът се добавя в контейнера за намерени обекти.

```

obstacle.fovCover = CAMERA_HORIZONTAL_FOV_ANGLE * 2 * (float)matchInfo[1]
[curIndex][1] / (float)resX;

obstacle.radius = obstacle.directDistance * sin(obstacle.fovCover / 2);

if (obstacle.radius < 0.01)
{
    continue;
}

obstacle.matchIndex = matchIndex;

obstacles.push_back(obstacle);

```

След като всички препятствия са определени и филтрирани, идва ред на проверката дали някой от тях лежи на правата на движение и по този начин блокира пътя на робота.

За целта обектите първо се сортират по разстоянието си до правата на движение посредством дефинираното сравнение между две препятствия.

```

// Compare two obstacles by their distance to the line the robot straight
walks on.
static bool operator<(const Obstacle& left, const Obstacle& right)

```

```

    {
        float leftObstaclePathLineDistance = left.directDistance *
sin(abs(left.angle)) - left.radius;
        if (leftObstaclePathLineDistance < 0)
        {
            leftObstaclePathLineDistance = 0;
        }

        float rightObstaclePathLineDistance = right.directDistance *
sin(abs(right.angle)) - right.radius;
        if (rightObstaclePathLineDistance < 0)
        {
            rightObstaclePathLineDistance = 0;
        }

        return leftObstaclePathLineDistance < rightObstaclePathLineDistance;
    }
}

```

След като препятствията са сортирани, е достатъчно да се намери първото такова, което не е по-далеч от целевата поза, и да се провери дали то блокира пътя. Това става в помощната функция `isPathBlocked`, в която `ROBOT_CIRCLE_RADIUS` има за стойност 0,2 м.

```

static
bool
isPathBlocked(
    const std::vector<Obstacle>& obstacles,
    const AL::Math::Pose2D& currentWorldPose,
    const AL::Math::Pose2D& targetWorldPose)
{
    float distanceToTarget = currentWorldPose.distance(targetWorldPose);

    for (std::vector<Obstacle>::const_iterator citer = obstacles.begin(),
end = obstacles.end(); citer != end; ++citer)
    {
        if (distanceToTarget + ROBOT_CIRCLE_RADIUS + citer->radius < citer-
>projectionDistance)
        {
            // The obstacles is too far away, therefore ignore it.
            continue;
        }

        return citer->directDistance * sin(abs(citer->angle)) < citer-
>radius + ROBOT_CIRCLE_RADIUS;
    }

    return false;    // There are no obstacles that are not too far away.
}

```

Ако не бъде намерено блокиращо пътя препятствия, то нищо повече не се прави и методът излиза. Ако обаче се открие такова препятствие, тогава се прави прекъсване на движението.

```

std::sort(obstacles.begin(), obstacles.end());

if (!isPathBlocked(obstacles, worldEventPose, worldTargetPose))
{
    return; // There are no blocking obstacles, so do nothing.
}

// There are blocking obstacles. Make an interruption.

```

```

{
    boost::lock_guard<boost::mutex> lock(this->statusMutex);
    this->status = STATUS_INTERRUPT;
}
// No need for now to notify any other thread.

// Abort the current compass move command.
bool result = this->compassProxy->moveTo(0, 0, 0);

```

С това приключва логиката на компонента за откриване на препятствия. Ако методът `onDeviation` все още не е излязъл, това означава че има блокиращо препятствие и е ред на компонента за заобикаляне на препятствия да се опита да намери заобиколен път.

### 6.1.7. Компонент за заобикаляне на препятствия

Компонентът за заобикаляне на препятствия е реално продължение на работата на компонента за откриване на препятствия във функцията за обратна връзка `onDeviation`. Затова и той получава намерените препятствия като просто използва вече дефинирания и напълнен с такива контейнер `obstacles` от тип `std::vector<Obstacle>`. Компонентът се опитва да намери междинна поза и в зависимост от това дали успява или не, връща различен резултат. Накрая нотифицира всички чакащи на условната променлива `statusCondvar` нишки, че е приключил работа. Такава нишка би трябвало да има в слоя, задаващ целевите пози, която след прекъсването на движението чака резултатите относно междинната поза.

```

// Find an alternative path, going around the obstacle.
AL::Math::Pose2D worldTransitPose;
if (findTransitPose(obstacles, worldEventPose, worldTargetPose,
worldTransitPose))
{
    boost::lock_guard<boost::mutex> lock(this->statusMutex);
    this->worldTransitPose = worldTransitPose;
    this->status = STATUS_REROUTE;
}
else
{
    boost::lock_guard<boost::mutex> lock(this->statusMutex);
    this->status = STATUS_NO_PATH;
}
this->statusCondvar.notify_all();

```

Повечето логика на този компонент се съдържа в помощната функция `findTransitPose`. Тя работи, като обхожда всички по-рано намерени препятствия на няколко етапа, като на всеки етап се търси решение с отклонение вляво и вдясно от правата на движение.

```

do
{
    retestSolution[LEFT] = false;
    retestSolution[RIGHT] = false;

    /* Retest solution or find new one */

}

```

```
while (retestSolution[LEFT] || retestSolution[RIGHT]);
```

Решението за всяка от двете страни представлява двойка ъгли, които са отклонението от правата на движение, което роботът трябва да направи, за да заобиколи успешно препятствията. Първият от двойката ъгли е отклонението встрани, ляво или дясно, спрямо правата, гледано от текущата поза към целевата поза. Вторият е ъгълът, който роботът ще сключи с правата, когато отново промени посоката си към целевата поза. Той обаче се разглежда като отклонение встрани от правата, гледано в обратната посока - от целевата поза към текущата поза. И понеже знакът на първия ъгъл определя дали роботът ще тръгне наляво или надясно, то вторият ъгъл може винаги да се разглежда с положителен знак.

```
float solutionCurrentAngle[2] = { 0, 0 };
float solutionTargetAngle[2] = { 0, 0 };
```

От тук нататък търсенето на решение за всяка от двете страни става по идентичен начин и кодът за това е общ, като само на някои места се взема предвид, че първият ъгъл на решението е положителен или отрицателен. Поради това на места ъгълът се взема по абсолютна стойност, а на други сравненията се правят в зависимост от знака на ъгъла. Нататък в изложението ще се представи само търсенето на решение с отклонение в лявата страна на правата, като търсенето на решение с отклонение в дясната страна на правата става по аналогичен начин.

Търсенето на решение започва от нулеви ъгли, като постепенно те се увеличават по абсолютна стойност, т.е. те се отварят повече. Търсейки решение по този начин се гарантира, че ако се намери такова, за всяка от страните поотделно, то ще е оптимално по отношение на отклонението, което роботът ще трябва да направи. Все пак това отваряне на ъглите има ограничение – то трябва да е в зрителното поле на робота.

```
if ((abs(solutionTargetAngle[side]) > CAMERA_HORIZONTAL_FOV_ANGLE / 2) ||
    (solutionTargetAngle[side] > AL::Math::PI / 2))
{
    // As these values are not allowed, there is no solution for this side.
    retestSolution[side] = false;
}
```

Когато текущото решение се промени, т.е. някой от двета ъгъла се увеличи, цялото решение трябва да се тества наново за колизии с всички препятствия, защото няма гаранция, че предварително разгледано препятствия няма да попречи на движението, използващо новонамерените ъгли. Целият процес на търсене на решение се повтаря, докато не се наложи промяна в намереното решение (т.е. то се валидира), или докато ъглите на решението не излязат от допустимите граници или надвишат ъглите на валидирано вече решение за другата страна.

На всяка итерация на цикъла за валидиране на решение се обхождат всички препятствия и за всяко едно от тях се проверяват редица неща. Първо, ако препятствието е твърде далеч напред, то се пропуска. След това се

проверява дали то заема целевата поза и ако това е така, директно се връща резултат, че път до целта не съществува, понеже нейната позиция е заета от обект.

```
std::vector<Obstacle>::const_iterator citer = obstacles.begin(), end =
obstacles.end();
for ( ; citer != end; ++citer)
{
    if (distanceToTarget + ROBOT_CIRCLE_RADIUS + citer->radius < citer-
>projectionDistance)
    {
        // The obstacles is too far away, therefore ignore it.
        continue;
    }

    // Calculate the distance between the target pose and the obstacle
    center.
    float obstacleTargetDistance = cosineLaw(distanceToTarget, citer-
>directDistance, abs(citer->angle));
    if (ROBOT_CIRCLE_RADIUS + citer->radius >= obstacleTargetDistance)
    {
        // An obstacle occupies the target pose!
        return false;
    }

    /* Other obstacle analysis */
}
```

Следва проверка дали препятствието е от другата страна на правата на движение и ако е така, то отново не се разглежда. В този момент има две възможности – ъгълът за заобикаляне на препятствието от външната му страна (спрямо правата на движение) е по-малък или по-голям от решението за ъгъла при текущата поза.

Ако ъгълът е по-малък, следва проверка за това дали решението заобикаля препятствието пред или зад него. Ако е зад него, то се проверява дали решението го пресича и ако е така, решението за ъгъла при целевата поза се увеличава до съответната стойност. Ако го заобикаля пред него, се проверява дали решението го пресича и ако е така, отново се увеличава решението за ъгъла при целевата поза. Макар да звучат еднакво, тези две проверки са различни.

Ако ъгълът е по-голям, то първо се проверява дали решението не подминава препятствието, оставяйки го встрани (от външната си страна) и ако е така, нищо повече не се прави. В противен случай решението за ъгъла при текущата поза се увеличава и допълнително се прави проверка дали трябва да се увеличи и решението за ъгъла при целевата поза, така че то да заобиколи разглежданото препятствие от задната му страна.

При всяка една промяна на кое да е от решениета, то се маркира, че трябва да бъде валидирано отново, което ще означава поне още една итерация на целия цикъл за намиране на решение.

След като цикълът за намиране на решение приключи, отново се проверява дали е намерено решение и ако има повече от едно такова, се избира по-доброто, след което се изчислява междинната поза, която то определя. Ако няма решение, то няма и път до целевата поза.

```

if (!foundSolution[LEFT] && !foundSolution[RIGHT])
{
    // There is no solution on either side, so there is no available path.
    return false;
}

// There is a solution and therefore an available path.
size_t pathSide;
if (foundSolution[LEFT] && foundSolution[RIGHT])
{
    // There are solutions on both sides, choose the better one.
    pathSide = solutionCurrentAngle[LEFT] + solutionTargetAngle[LEFT] <
               solutionCurrentAngle[RIGHT] + solutionTargetAngle[RIGHT];
}
else
{
    // There is only one found solution, so use it.
    pathSide = foundSolution[LEFT] ? LEFT : RIGHT;
}

// Calculate how far the transit point is.
float distanceToTransit = distanceToTarget *
sin(solutionTargetAngle[pathSide]) /
sin(solutionTargetAngle[pathSide] +
abs(solutionCurrentAngle[pathSide]));

worldTransitPose.theta = AL::Math::modulo2PI(worldCurrentPose.theta +
solutionCurrentAngle[pathSide]);
worldTransitPose.x = worldCurrentPose.x + distanceToTransit *
cos(worldTransitPose.theta);
worldTransitPose.y = worldCurrentPose.y + distanceToTransit *
sin(worldTransitPose.theta);

return true;

```

### 6.1.8. Компонент, използващ сонара

Работата на компонента, използващ сонара, е много прости. Ако той засече обект на достатъчно близка дистанция, той ще спре движението на робота. За такива са приети обекти на разстояние по-малко от 30 см, като не се прави разлика дали те са вляво или вдясно. Допълнително се проверява дали засечените обекти не са на разстояние, по-голямо от това до целевата поза. Последното ще позволи на робота да достига позиции, които са близо до обекти, като например стени или мебели.

```

void vcoa::onSonarDetection(const std::string& eventName, float
objectDistance, std::string& /* subscriberId */)
{
    if (objectDistance > 0.3)
    {
        return;
    }

    AL::Math::Pose2D worldCurPose, worldTargetPose;

    if (this->getCurrentWorldPose(worldCurPose))
    {
        boost::unique_lock<boost::mutex> lock(this->statusMutex);

```

```

        worldTargetPose = (this->status == STATUS_GOAL) ? this-
>worldGoalPose : this->worldTransitPose;

        float targetDistance = worldTargetPose.distance(worldCurPose);
        if (objectDistance > targetDistance)
        {
            return;
        }

    }

    // There are obstacles along the path. Stop the movement.
    {
        boost::lock_guard<boost::mutex> lock(this->statusMutex);
        this->sonarDetectedObstacle = true;
    }

    // Abort the current compass move command.
    this->compassProxy->moveTo(0, 0, 0);
}

```

## 6.2. Интеграция

Интеграцията на модула за откриване и заобикаляне на препятствия е лесна. Това важи, както за инсталацирането и използването, така и за внедряването в графичната среда за програмиране Choregraphe. До голяма степен това се дължи на простия му потребителски интерфейс, но така също и на възможностите за интеграция и простотата, с която това се прави, на платформата за разработване на модули на робота Nao. Това бе разгледано в глава 3.1.

### 6.2.1. Инсталiranе и използване

Модулът за откриване и заобикаляне на препятствия се инсталира лесно. Макар да няма направена програма инсталатор, той представлява само един файл. В зависимост от това дали модулът ще се използва като локален или като нелокален, това ще бъде динамично зареждана библиотека или изпълним файл, работещи съответно на робота или на друга машина, свързана по мрежата с него. До момента модулът е бил компилиран и тестван само като нелокален модул на Windows машина, но би трявало да работи по същия начин като локален или нелокален, компилиран за друга операционна система.

В проведените тестове, модулът се стартира на Windows машина със следния команден ред:

```
vcoa_d.exe --pip <ip address> --pport <port number>
```

Тук *vcoa\_d.exe* е името на изпълнимия файл с модула, а *--pip* и *--pport* са параметри съответно за IP (версия 4) адрес на робота, за който модулът ще работи, и порт, на който да се свърже. Това е всичко необходимо, за да може да се използва модулът за откриване и заобикаляне на препятствия на робота.

Самата употреба на модула става отново лесно – използваният модула създава посредник от общ вид за модула *vcoa*, след което през него прави заявки към *moveTo* метода на класа. Накрая резултатът може да се анализира.

```

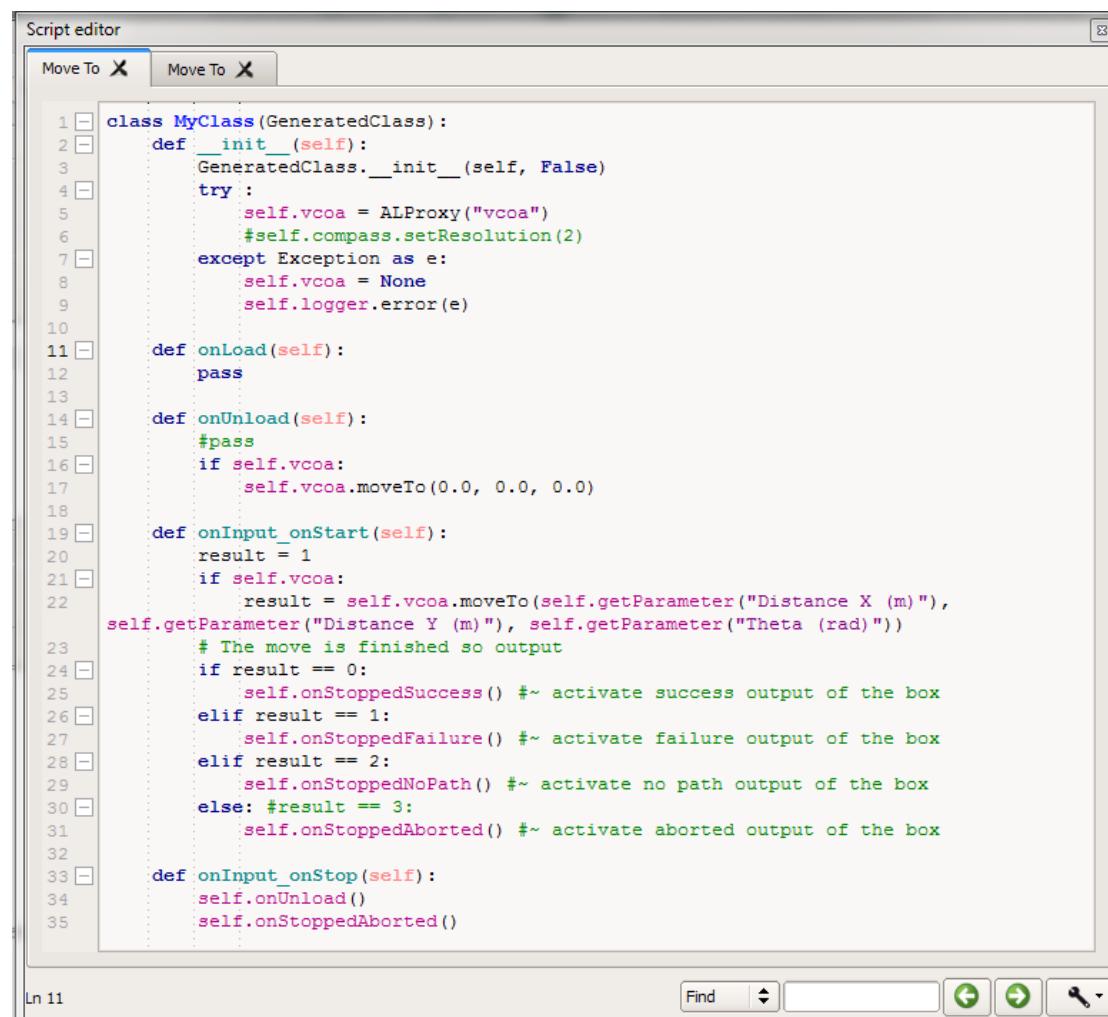
boost::shared_ptr<AL::ALBroker> broker =
AL::ALBroker::createBroker("MyBroker", "", 0, robotIP, port);
boost::shared_ptr<AL::ALProxy> testProxy =
boost::shared_ptr<AL::ALProxy>(new AL::ALProxy(broker, "vcoa"));

unsigned int result;
result = testProxy->call<unsigned int>("moveTo", x, y, theta);

```

## 6.2.2. Интеграция със средата за програмиране Choregraphe

Хубаво е модулът vcoa да бъде достъпен за употреба и в графичната среда за програмиране Choregraphe. Макар той да няма собствена кутия (*box*), стандартната кутия на визуалния компас лесно може да бъде преобразувана, за да използва модула за откриване и заобикаляне на препятствия. *Move To* кутията в него се модифицира, добавяйки двата възможни изхода за липса на път и отмяна на команда. Кодът в нея се променя, така че посредникът да бъде към модула vcoa и резултатът на извикването към *moveTo* да бъде обработен по правилен начин.



The screenshot shows the 'Script editor' window in the Choregraphe software. The title bar says 'Script editor'. Below it is a toolbar with two buttons: 'Move To X' and 'Move To X'. The main area contains a script with line numbers from 1 to 35. The script defines a class 'MyClass' that inherits from 'GeneratedClass'. It includes methods for initialization, loading, unloading, and handling input events like start and stop. The code uses Python syntax and interacts with the vcoa module via ALProxy.

```

1  class MyClass(GeneratedClass):
2      def __init__(self):
3          GeneratedClass.__init__(self, False)
4          try:
5              self.vcoa = ALProxy("vcoa")
6              #self.compass.setResolution(2)
7          except Exception as e:
8              self.vcoa = None
9              self.logger.error(e)
10
11     def onLoad(self):
12         pass
13
14     def onUnload(self):
15         #pass
16         if self.vcoa:
17             self.vcoa.moveTo(0.0, 0.0, 0.0)
18
19     def onInput_onStart(self):
20         result = 1
21         if self.vcoa:
22             result = self.vcoa.moveTo(self.getParameter("Distance X (m)",
23                                         self.getParameter("Distance Y (m)", self.getParameter("Theta (rad)")))
24             # The move is finished so output
25             if result == 0:
26                 self.onStoppedSuccess() #~ activate success output of the box
27             elif result == 1:
28                 self.onStoppedFailure() #~ activate failure output of the box
29             elif result == 2:
30                 self.onStoppedNoPath() #~ activate no path output of the box
31             else: #result == 3:
32                 self.onStoppedAborted() #~ activate aborted output of the box
33
34     def onInput_onStop(self):
35         self.onUnload()
36         self.onStoppedAborted()

```

Снимка 3: Кодът на vcoa кутията

Промененият код е показан на Снимка 3 (направена от екран, за да се запазят важните за синтаксиса на езика Python табулатии).

### 6.3. Тестване

По време на работата с робота се наблюдаваха много неточности в предоставените от него данни и действията му. По-специфично, одометрията му не работи особено точно и почти винаги има значими отклонения в движенията. За да се провери това, на стандартния модул за движение бе зададено простото движение от изминаване на кратки разстояния по права линия – 40 см, 60 см, 1 м, 1,5 м. И двата робота изминаха приблизително правилно разстояние напред, но имаха забележимо отклонение встрани – и при двата наляво, като при единия, синия, то бе по-голямо. Очаквано с увеличаването на подаденото разстояние, отклонението нараства, достигайки почти до обръщане на лицето на 90 градуса от по-неточния робот.

Предложениет от Aldebaran модул за коригиране на отклонение при движение е визуалният компас, който е и важна част от разработвания от настоящата дипломна работа модул. Затова аналогични тестове бяха проведени и с него. При всички разстояния действително имаше коригиране на отклонението встрани, като оранжевият робот почти не се отклоняваше от правата, докато синият го правеше, но значително по-малко, отколкото при употребата на стандартния модул за движение. Въпреки това бе наблюдавано, че при употребата на визуалния компас роботите правеха видимо по-малки крачки и изминаха значително по-малко разстояние от подаденото им. Във всички случаи изминатото бе малко повече от приблизително 2/3 от подаденото разстояние, като това варира донякъде в зависимост от фоновите особености и най-вече осветлението. Резултатите за движение напред на 171 см с визуалния компас, извършени от синия робот, са представени в Таблица 1. Трябва да се отбележи, че освен отклонение вляво, роботът правеше и завъртане наляво, от около приблизително 20-30 градуса.

опит	изминато разстояние	измината част	отклонение вляво
1	116	0,68	10
2	121	0,71	4
3	119	0,7	8
4	120	0,7	8
5	120	0,7	15
6	119	0,7	10
7	118	0,69	6
8	122	0,71	12
9	117	0,68	10
10	123	0,72	8
средно	119,5	0,7	9,1

Таблица 1: Реално изминато разстояние, с визуален компас, при зададено движение напред от 171 см

Аналогичен тест бе проведен при употребата на модула за откриване и заобикаляне на препятствия, без такива да са налични на пътя на робота.

Резултатите от него, отново проведени със синия робот, са представени в Таблица 2, като бележката е, че след като извърши аналогично движение на това само с визуалния компас, роботът се завърта допълнително наляво. В резултат от това, центърът му е изместен малко по-напред, но и по-наляво, и поважното – крайното му завъртане е вече значително по-голямо, достигащо повече от 45 градуса.

опит	изминато разстояние	измината част	отклонение вляво
1	132	0,77	16
2	127	0,74	15
3	126	0,74	12
4	128	0,75	11
5	125	0,73	14
6	124	0,73	16
7	127	0,74	11
8	124	0,73	0
9	128	0,75	1
10	128	0,75	19
средно	126,9	0,74	11,5

*Таблица 2: Реално изминато разстояние, с модула за откриване и заобикаляне на препятствия, при зададено движение напред от 171 см*

Причината за това допълнително завъртане бе, че получените от датчиците на робота данни показват на модула за откриване и заобикаляне на препятствия, че роботът се е завъртял надясно, макар и реално това да не е така. За да компенсира отклонението, модулът задава завъртане в обратната посока – наляво. Въпросните данни са взети посредством помощната функция `getCurrentWorldPose`, която бе представена в глава 6.1.3.

След теста без препятствия на пътя, бе проведен тест с едно препятствие. То бе с размери 15 см на 6 см и бе на разстояние от 130 см от началната позиция на робота. За съжаление, модулът за откриване и заобикаляне на препятствия не успя да се справи със задачата си и в повечето случаи игнорираше препятствието, минавайки през него. В други случаи пък се отклоняваше встрани, очевидно неправилно. Все пак, имаше и случаи, в които роботът отчиташе препятствието и го заобикаляше. Това се удава само на оранжевия робот.

Предвид, че тестът с едно препятствие бе неуспешен, усилията бяха насочени към подобряване на представянето на роботите на него. Съответно не бяха проведени по-сложни тестове – такива с повече препятствия и на по-големи разстояния.

Допълнително затруднение бе, че по-добре представящият се робот, оранжевият, получи повреда – вентилаторът в главата му спря да работи, което пък доведе до бързото прегряване на процесора. Това направи работата с този робот почти невъзможна.

## **6.4. Анализ на резултатите от тестването**

Проведените тестове с предоставените от Aldebaran модули за движение показват, че одометрията на робота Nao не е прецизна. Дори когато се използва визуална корекция, движенията не са точни. Представянето допълнително се влошава в следствие на износване и неподходяща настилка, слабо осветление и липса на особености в околната среда. Възможно е да има и други влияещи фактори.

Общото впечатление от работата с датчиците на робота, осигуряващи одометрията му, е, че те не работят задоволително добре. И докато за невъзпрепятствано придвижване това може да не е толкова голям проблем, когато знанието за изминатото разстояние е ключов елемент от алгоритмите за откриване и заобикаляне на препятствия, всяка неточност в получените данни е с катастрофални последствия за представянето на разработвания в настоящата дипломна работа модул. Повлияни са изчисленията, както за местоположението и размерите на препятствията, така и за самото движение на робота сред тях. Показателно за това е, че помощната функция `getCurrentWorldPose`, която се извиква от всички функции за движение и откриване на препятствия, извлича резултата си имено от тези датчици, без да взима предвид визуалната информация от визуалния компас. За съжаление, този факт не бе описан в документацията на робота Nao, което попречи на навременното взимане на мерки за справяне с този проблем. Допълнителен проблем е, че дори достъпването на тези данни не става по един надежден начин – в голяма част от случаите, функцията не успява, понеже получава от модула `AL::ALMemory` никакви или некоректно форматирани данни.

Друг проблем, вече на използвания алгоритъм, е, че особеностите, получени от визуалния компас, в този си сувор вид, не са достатъчни, за да се определят препятствията на пътя на робота. Налага се въвеждането на минимален размер на обектите, за да се разглеждат те въобще, което обаче може да елиминира истински препятствия (границата от 1 см бе експериментално определена). Изглежда местоположението им в зрителното поле на робота е достатъчно, за да може да се коригира движението му, до някаква степен, но не е надеждно и достатъчно, за да се откриват препятствия.

Положителният резултат от проведеното тестване е, че кодът на реализацията е сравнително изчистен от грешки. В това число влиза логиката на командния слой и слоя, задаващ целевите пози, както и на целия механизъм на прекъсвания поради блокиращи пътя препятствия, намиране на междинна поза (макар и по погрешен начин) и последващото пренасочване на робота. Мнението на автора е, че този код, занимаващ се с управлението на робота, може да бъде преизползван в комбинация с едно по-качествено и надеждно придвижване и намиране на препятствия.

# Глава 7. Заключение

## 7.1. Обобщение на изпълнението на началните цели и претенциите за оригинални резултати

В рамките на настоящата дипломна работа първоначално поставените цели са изпълнени, макар демонстрираните резултати от имплементацията да не са задоволителни. Проучена бе платформата за програмиране на робота Nao и начинът за създаване, интегриране и използване на нови модули за него. Разгледани бяха теоретични и съществуващи подходи за откриване на препятствия, като бе избран такъв, който има своите достойнства – простота и употреба само на наличните уреди. Реализиран бе модул за робота Nao, който открива препятствия на пътя на робота и дава инструкции за тяхното заобикаляне. Бяха направени тестове върху него и други стандартни модули на робота, като анализът на резултатите и на двете показва, че тяхното представяне е нездадоволително. Показано бе и как разработеният модул може да бъде интегриран с графичната среда за програмиране Choregraphe.

Сред намерените материали за проучвания на проблема с откриване и заобикаляне на препятствия от робота Nao не бе намерен такъв, който да възприема подхода, разгледан в настоящата дипломна работа. Затова резултатите, макар и не добри, представляват интерес, защото демонстрират, че показанията на одометричните датчици на робота и сурвите данни от неговата камера са недостатъчни за постигането на поставената цел. Абсолютно наложителни са по-задълбочен анализ на данните и методи за неутрализиране на шума в замерванията.

Позитивно следствие от дипломната работа е, че авторът ѝ придоби ценен опит и знания в сферата на роботиката. Те биха му били много полезни в евентуалната му бъдеща работа, свързана с физически роботи.

## 7.2. Насоки за бъдещо развитие и усъвършенстване

Ако се вземе решение реализираният модул за откриване и заобикаляне на препятствия да бъде доразработван, има някои насоки за неговото усъвършенстване. Първото и най-важно нещо е да се направи по-точно намиране на изминатото от робота разстояние между референтната и текущата пози. За тази цел може да са необходими прецизна калибрация на камерата и управлението на моторите на робота.

Следваща възможност е да се реализира начин за извлечане на изображенията, които използва робота, за да могат да се разгледат с цел по-лесно и нагледно анализиране на намерените от него съответствия. Начален опит за постигането на това бе направен и в процеса на разработване на настоящата работа, но той бе неефективен по отношение на запазването на изображенията. Това е свързано с необходимостта от доста време за трансфера им към отдалечената машина, на която работи имплементираният модул, като това вероятно може да бъде елиминирано, ако модулът бъде направен да работи като локален на робота.

Друга възможна насока за развитие е прилагането на по-добър метод за намиране на входните данни на използвания паралакс метод. Това може да включва елиминиране на шума, породен от малките, но съществуващи, разклащания на робота по време на ходене. Възможен подход за това е чрез намиране на център на гравитация на особеностите в референтното и текущото изображения и използването на него за отправна точка в изчисляването на ъглите  $\alpha$  и  $\beta$  на паралакс метода, вместо централната точка на всяко едно от двете изображения. Това може да подобри точността на намерените ъгли, понеже ще става по отношение на проследена между двете изображения точка, вместо да се разчита на това, че роботът действително гледа право напред и при двете изображения.

Чисто имплементационно, модулът може да се подобри чрез въвеждане на по-ефективна система за логове, възможност за конфигурация по време на изпълнение и въвеждане на специален режим за откриване на грешки (*debug mode*), който да събира повече информация за работата на модула с цел нейното анализиране.

# Използвана литература

- [1] ALSonar. (б.д.). Aldebaran Robotics. Достъпено онлайн на 20.10.2015, <http://doc.aldebaran.com/2-1/naoqi/sensors/alsonar.html>
- [2] Stereopsis. (б.д.). Wikimedia Foundation. Достъпено онлайн на 20.10.2015, <https://en.wikipedia.org/wiki/Stereopsis>
- [3] Structure from motion. (б.д.). Wikimedia Foundation. Достъпено онлайн на 20.10.2015, [https://en.wikipedia.org/wiki/Structure\\_from\\_motion](https://en.wikipedia.org/wiki/Structure_from_motion)
- [4] How a Roomba robot vacuum cleaner works. YouTube (б.д.). Достъпено онлайн на 20.10.2015, <https://www.youtube.com/watch?v=uCWeG3p5KJA>
- [5] Kooijman, Laan, Verschoor, & Wiggers. (04.02.2013). NAVIGATE. *Faculty of Science, University of Amsterdam.*
- [6] Fojtu, S., Havlena, M., & Pajdla, T. (б.д.). *Nao Robot Localization and Navigation Using Fusion of Odometry and Visual Sensor Data*. Prague: Center for Machine Perception, Department of Cybernetics, FEE, CTU in Prague.
- [7] George, L., & Mazel, A. (2013). *Humanoid Robot Indoor Navigation Based on 2D Bar Codes: Application to the NAO Robot*. Aldebaran Robotics A-Lab. работен вариант
- [8] Wei, C., Xu, J., Wang, C., Wiggers, P., & Hindriks, K. (б.д.). *An Approach to Navigation for the Humanoid Robot Nao in Domestic Environments*. Delft: EEMCS, Delft University of Technology.
- [9] Hornung, A., Osswald, S., Maier, D., & Bennewitz, M. (2014). *MONTE CARLO LOCALIZATION FOR HUMANOID ROBOT NAVIGATION IN COMPLEX INDOOR ENVIRONMENTS*. Freiburg: Department of Computer Science, University of Freiburg.
- [10] Maier, D., Bennewitz, M., & Stachniss, C. (б.д.). *Self-supervised Obstacle Detection for Humanoid Navigation Using Monocular Vision and Sparse Laser Data*. Freiburg: Department of Computer Science, University of Freiburg.
- [11] Fojtu, S., Prusa, D., & Pajdla, T. (2010). *Towards Robot Localization and Obstacle Avoidance from Nao Camera*. Prague: Center for Machine Perception, Department of Cybernetics, Faculty of Electrical Engineering, Czech Technical University in Prague.
- [12] ALVisualCompass. (б.д.). Aldebaran Robotics. Достъпено онлайн на 20.10.2015, <http://doc.aldebaran.com/2-1/naoqi/vision/alvisualcompass.html>

- [13] Nao (robot). (б.д.). Wikimedia Foundation. Достъпено онлайн на 20.10.2015, [https://en.wikipedia.org/wiki/Nao\\_\(robot\)](https://en.wikipedia.org/wiki/Nao_(robot))
- [14] NAO – Construction. (б.д.). Aldebaran Robotics. Достъпено онлайн на 20.10.2015, [http://doc.aldebaran.com/2-1/family/robots/dimensions\\_robot.html](http://doc.aldebaran.com/2-1/family/robots/dimensions_robot.html)
- [15] Retrieving software. (б.д.). Aldebaran Robotics. Достъпено онлайн на 22.10.2015, [http://doc.aldebaran.com/2-1/dev/community\\_software.html](http://doc.aldebaran.com/2-1/dev/community_software.html)
- [16] Extending NAO API - Creating a new module. (б.д.). Aldebaran Robotics. Достъпено онлайн на 20.10.2015, [http://doc.aldebaran.com/2-1/dev/cpp/tutos/create\\_module.html](http://doc.aldebaran.com/2-1/dev/cpp/tutos/create_module.html)
- [17] Key concepts. (б.д.). Aldebaran Robotics. Достъпено онлайн на 20.10.2015, <http://doc.aldebaran.com/2-1/dev/naoqi/index.html>
- [18] Rosten, E., & Drummond, T. (2006.). *Machine learning for high-speed corner detection*. Department of Engineering, Cambridge University.
- [19] Feature Detection and Description. (б.д.). opencv dev team. Достъпено онлайн на 20.10.2015, [http://docs.opencv.org/modules/features2d/doc/feature\\_detection\\_and\\_description.html#fastOpenCV](http://docs.opencv.org/modules/features2d/doc/feature_detection_and_description.html#fastOpenCV)
- [20] Fast Approximate Nearest Neighbor Search. (б.д.). opencv dev team. Достъпено онлайн на 20.10.2015, [http://docs.opencv.org/modules/flann/doc/flann\\_fast\\_approximate\\_nearest\\_neighbor\\_search.html](http://docs.opencv.org/modules/flann/doc/flann_fast_approximate_nearest_neighbor_search.html)
- [21] RANSAC. (б.д.). Wikimedia Foundation. Достъпено онлайн на 20.10.2015, <https://en.wikipedia.org/wiki/RANSAC>
- [22] Perspective (visual). (б.д.). Wikimedia Foundation. Достъпено онлайн на 20.10.2015, [https://en.wikipedia.org/wiki/Perspective\\_\(visual\)](https://en.wikipedia.org/wiki/Perspective_(visual))
- [23] Example: HelloWorld module. (б.д.). Aldebaran Robotics. Достъпено онлайн на 21.10.2015, <http://doc.aldebaran.com/2-1/dev/cpp/examples/core/helloworld/example.html>
- [24] C++ SDK Installation. (б.д.). Aldebaran Robotics. Достъпено онлайн на 22.10.2015, [http://doc.aldebaran.com/2-1/dev/cpp/install\\_guide.html](http://doc.aldebaran.com/2-1/dev/cpp/install_guide.html)

# **Фигури, таблици и снимки**

## **Списък на фигурите**

Фигура 1: Ъгъл в равнината към обект.....	20
Фигура 2: Намиране на разстояние до обект.....	22
Фигура 3: Разстояние на обект до правата на движение.....	24
Фигура 4: Ъгли на заобикаляне на препятствие.....	26
Фигура 5: Различни позиции на препятствия.....	27
Фигура 6: Архитектура на модула.....	30
Фигура 7: Команден слой.....	31
Фигура 8: Слой, задаващ целевите пози.....	32
Фигура 9: Слой, извършващ движенията.....	34

## **Списък на таблиците**

Таблица 1: Реално изминато разстояние, с визуален компас, при зададено движение напред от 171 см.....	55
Таблица 2: Реално изминато разстояние, с модула за откриване и заобикаляне на препятствия, при зададено движение напред от 171 см.....	56

## **Списък на снимките**

Снимка 1: Хуманоиден робот Nao.....	4
Снимка 2: Nao роботите ESSYIKA (оранжев) и ESSYIKO (син).....	12
Снимка 3: Кодът на всяка кутията.....	54