UNIVERSITÄT
KOBLENZ · LANDAU
Fachbereich 4: Informatik

# Collision-free path planning for a robotic arm in RoboCup @Home

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Kevin Read

**Betreuer:** Dipl.-Inform. D. Gossow, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
**Erstgutachter:** Dipl.-Inform. D. Gossow, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau
**Zweitgutachter:** Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik, Universität Koblenz-Landau

Koblenz, im Juni 2010

## Kurzfassung

Das Ziel dieser Studienarbeit ist es, einen Roboterarm in einen bestehenden Software-Stack zu integrieren, damit ein darauf basierender Roboter beim Wettbewerb RoboCup @Home teilnehmen kann.

Der Haushaltsroboter Lisa (Lisa Is a Service Android) muss für den @Home-Wettbewerb unter anderem Gegenstände aus Regalen entnehmen und an Personen weiterreichen. Bisher war dafür nur ein Gripper, also ein an der mobilen Plattform in Bodennähe angebrachter »Zwicker« vorhanden. Nun steht dem Roboter ein »Katana Linux Robot« der Schweizer Firma Neuronics zur Verfügung, ein Roboter in Form eines Arms. Dieser wird auf LISA montiert und nimmt über verschiedene Schnittstellen Befehle entgegen. Er besteht aus sechs Gliedern mit entsprechend vielen Freiheitsgraden. Im Robbie-Softwarestack muss ein Treiber für diesen Arm integriert und eine Pfadplanung erstellt werden. Letztere soll bei der Bewegung des Arms sowohl Kollisionen mit Hindernissen vermeiden als auch natürlich wirkende Bewegungsabläufe erstellen.



## Abstract

The goal of this minor thesis is to integrate a robotic arm into an existing robotics software. A robot built on top of this stack should be able to participate successfully RoboCup @Home league.

The robot Lisa (Lisa is a service android) needs to manipulate objects, lifting them from shelves or handing them to people. Up to now, the only possibility to do this was a small gripper attached to the robot platform. A »Katana Linux Robot« of Swiss manufacturer Neuronics has been added to the robot for this thesis. This arm needs a driver software and path planner, so that the arm can reach its goal object »intelligently«, avoiding obstacles and creating smooth, natural motions.

## Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Vereinbarung der Arbeitsgruppe für Studien- und Abschlussarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts.

| | | |
|---|---|---|
| Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden. | ja ☐ | nein ☐ |
| Der Veröffentlichung dieser Arbeit im Internet stimme ich zu. | ja ☐ | nein ☐ |

Koblenz, den 27th July 2010

# Contents

# List of Tables

# List of Figures

# Chapter 1

# Introduction

In this minor thesis a path planner for a robotic arm and necessary driver infrastructure to interface the Katana M400 arm will be added to the robotics software stack developed at the Active Vision Working Group (AGAS) at the University of Koblenz-Landau. The goal is to participate in the RoboCup @Home league tests that require object manipulation.

This chapter will give an overview of the RoboCup @Home league with an eye on the challenges that involve object manipulation. Next, an overview of the robotic software stack used shall be presented. Finally the hardware platform »Lisa« and the robotic arm shall be introduced.

## 1.1 The RoboCup @Home league

RoboCup is an international robotics competition that aims to further research in robotics and artificial intelligence by »providing a standard problem where wide range of technologies can be integrated and examined, as well as being used for integrated project-oriented education.«[1] This standard problem was originally defined as a football match, building on the games well-defined rules, its high popularity and the multitude of technologies that can used as the foundation for a successful soccer team, as in multi-agent collaboration, autonomy, sensor fusion and other research topics. RoboCup events also house conferences and workshops.

The competition was split into various leagues that concentrate on different research topics or sub-problems of soccer. In 2001 the first non-soccer league »RoboCup Rescue« was added to the competition, which focuses on the use of robots in disaster recovery. In 2006 a new league was created to research the use of robots in a household environment, where they can help with everyday needs and tasks. This league was named »RoboCup @Home«, and has since attracted very

[1] source: `http://www.robocup.org/`

large interest. At this years RoboCup 2010 in Singapore, the @Home competition was attended by twenty-four teams from around the globe, which necessitated two games being executed in parallel to conserve time.

Recurring topics in RoboCup @Home are object, speech and face recognition, path and motion planning and Human-Computer Interaction. This is reflected in the competition challenges, called games, that participants have to absolve. An example would be the game »Shopping Mall«, where the robot goes into said structure and tries to correctly identify and then collect items from shelves. This game also highlights the necessity for object manipulation.

## 1.2 The software stack

The Active Vision Working Group of the University of Koblenz-Landau has been involved with RoboCup since 2006. The software stack was originally developed for student projects and research into robotic topics, but could be adapted to the needs of the RoboCup teams. It is written in C++ and has been designed with extensibility in mind and has been re-used and extended since the first RoboCup. Initially used in the Rescue competition, it serves as the basis for the @Home team now, too.

The stack is modelled after the mediator pattern, utilizing a central message queue and a subscription system. Application logic is partitioned into modules that subscribe to certain messages and can send messages of their own. Modules can be proactive (they wake up in regular intervals and on message reception) or reactive (they wake up only on message reception). Sensors and actuators reside within driver modules, data processing is done in Worker modules, which can embed external libraries.

## 1.3 The robot Lisa

Lisa is a recursive acronym that stands for Lisa Is a Service Android and is the platform of the @Home team »Homer«. It consists of a MobileRobots Robotics Pioneer P3AT[2] platform, a four-wheeled platform with a front-mounted gripper. On top of this platform the custom-made framework designed and built by Centre of Excellence of the Chamber of Crafts in Koblenz[GWB+10] is mounted, which carries most sensors and the controlling notebook. The framework consists of a solid base that covers the platform, and an elongated tower-like structure toward the read end of the platform that carries a pan-tilt unit with a sensor array.

[2] `http://www.activerobots.com`


**Figure 1.1:** Lisa showing of the robotic arm during the RoboCup @Home 2010 Open Challenge

The framework houses two laser range finders. One is mounted directly above the platform and is used for navigation purposes. The second LRF is part of said sensor array, using the flexible positioning for 3-dimensional scanning. Other sensors found in the array are a camera and a time-of-flight camera for face or gesture detection and recognition and a microphone. There is a LCD touch screen embedded into the tower part of the framework that typically shows Lisas »face«, an iconic human face that can show different emotions and moves its lips during speech output through the loudspeaker next to the screen.

The arm is mounted on the surface in front of the tower which slopes slightly towards the front face of the framework, so that the robotic arm can reach down into the working area of the platforms gripper. Both sides are free of obstacles, and the tower slants backwards, away from the arm, to maximize its available room. The robot is depicted in figure 1.1.

## 1.4 The Katana 450 arm robot

The Neuronics Katana 450[3] is a standalone, segmented arm-shaped robot that consists of a controller box running an embedded Linux system and three joined arm segments with a maximum operation radius of 517mm ending in a configurable

[3] `http://www.neuronics.ch/`

appendage, in our case a two-pronged gripper. There are six motors built into the robot, leading to six degrees of freedom (including one in the gripper). The robot can be interfaced via USB or Ethernet. More information on the robot can be found at `http://www.neuronics.ch/cms_en/web/index.php?id=244&s=katana`.

# Chapter 2

# The state of segmented robot motion planning

Motion planning for robots is not a young field of research, and there are many well-understood and documented approaches in use today. A good overview can be found in [CLH+05]. Most general-purpose motion planning algorithms apply to segmented robots, although their high dimensionality makes some approaches less feasible or even infeasible.

I will present the state of robot motion planning algorithms with a focus on those that work well for robots with at least six degrees of freedom, which is a very common configuration for segmented robots.

## 2.1 Potential functions

A well-understood group of algorithms are potential functions. They are not suited for high-dimensional search problems. Intuitively, if a robot where a positively charged particle, potential functions employ gradients that act as a negative force to attract this particle to the goal. Obstacles act as positively charged forces to repel the robot. »The combination of repulsive and attractive forces hopefully directs the robot from the start location to the goal location while avoiding obstacles.«[CLH+05, p. 77]

A logical approach to this problem would be gradient descent: »Starting at the initial configuration, take a small step in the direction opposite the gradient. This gives a new configuration, and the process is repeated until the gradient is zero.« [CLH+05, p. 84] To calculate the repelling force from obstacles, the distance to these must be established. [CLH+05, p. 86] introduce the Brushfire algorithm as an efficient algorithm to compute this distance. Intuitively, a map is created in form of a grid of pixels. All non-occluded pixels are initialized with zero, all other

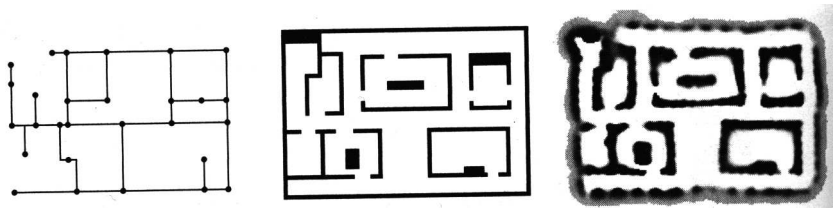

**Figure 2.1:** Different kinds of roadmaps: topological, geometric and grid-based. Source: [CLH+05, p.108]

pixels with one. Now all zero-valued pixels that have a neighbour with a value of one are set to two. In the next step, all zero-valued pixels with a neighbouring pixel of two are set to three, and so on. When a four-point connectivity is used as basis for the neighbour search, the pixel value corresponds to the Manhattan distance to the next obstacle.

One problem of potential functions is that the search can easily end in nestled in a concave obstacle or a set of convex obstacles that are too close together, essentially becoming concave. Because the repelling force of the obstacles and the attractive force of the goal that lies behind the obstacle cancel themselves out, the search deadlocks. This is known as the local minimum problem as stated in [CLH+05, p.90], where the solution is given as »the wave-front planner [...] affords the simplest solution to the local minima problem, but can only be implemented in spaces that are represented as grids.« A good visualization of the wave-front planner is a wave front that starts from the goal and expands outwards, ending when it hits the start position. When the wave passes over a grid cell, it stores the distance to the goal in this cell. Occluded cells are avoided by the wave. The gradient descent then uses this distance in the cells as the gradient function. »The wave-front planner essentially forms a potential function on the grid which has one local minimum and thus is resolution complete«[CLH+05, p.91].

## 2.2 Roadmaps

Maps form the basis of many path planning approaches. They are used when incrementally building a map of the environment based on sensor information, or to pre-calculate planning information for an environment that can be reused often. [CLH+05, p.107] distinguishes between topological, geometric and grid-based maps.

Topological maps consist of a graph with the nodes representing sensor information and the edges showing possible transitions between these nodes. Geometric maps try to fit sensor observations into geometric shapes and note these on the map. Line segments or triangles are often used here. Grid-based maps note

the »likelihood that its corresponding portion of workspace or configuration space is occupied«[CLH+05, p.108] in each grid cell. These occupancy grids are used throughout the Robbie stack for navigation and mapping. Different kinds of maps for the same physical location are shown in figure 2.1.

Of particular interest for high-dimensional problems are roadmaps: Think of a map of railway stations, like a standard London underground map - a graph with the nodes representing phyiscal locations and the edges showing possible transitions between these locations. [CLH+05, p.108] on roadmaps:

> Robots use roadmaps in much the same way people use highway systems. Instead of planning every possible side-street path to a destination, people usually plan their path to a network of highways, then along the highway system, and finally from the highway to their destination. The bulk of motion occurs on the highway system, which brings the motorist from near the start to near the goal.

Many roadmap planners expect an explicit representation of obstacles in form of their geometry. Given explicit geometry, these planners are powerful. Examples of such planners are Visibility Maps[CLH+05, p.110], Deformation Retracts[CLH+05, p.117] or Piecewise Retracts[CLH+05, p.138]. The underlying approach is to compute valid paths from obstacle geometry. As our configuration space is at least four-dimensional, deriving the geometry of obstacles in the configuration space $Q$ from $R^3$ is not straightforward. Each point on the convex hull of the obstacle in $R^3$ might correspond to an unlimited number of points in $Q$, thereby rendering these planners impractical for our needs. [CLH+05, p.197].

[CLH+05, p.197] propose an alternative approach by sampling the configuration space and thereby generating a graph of configurations and interconnecting paths that lie in $Q_{free}$, the non-occluded part of configuration space. It is, in effect, a space-time tradeoff, investing computational power in advance to save it later. Research into this area launched after Canny showed that the generalized movers problem (in which a robot consists of a collection of polyhedra freely linked together at various vertices) was PSPACE-complete (polynomial complexity), so a less complex approach was needed.

The first such algorithm was PRM, the Probabilistic RoadMap planner. The assumption is that checking if a given configuration $q$ is in $Q_{free}$ or not is cheap. "It uses rather coarse sampling to obtain the nodes of the roadmap and very fine sampling to obtain the roadmap edges, which are free paths between node configurations." [CLH+05, p.198] To answer a query, only the connection from the start position to the roadmap and from there to the goal need to be checked against $Q_{free}$. The path through the roadmap from the start entry point to the goal exit point can be computed by doing graph search. The roadmap can be re-used for subsequent planning work as long as the environment does not change. The approach

can also be used for one-shot planning, where the starting and goal positions are also added to the map and the planning stops as soon as the goal is reached.

Special care has to be taken when choosing a sampling strategy. Using a random distribution of $q \in Q$ might produce an even coverage, but if the majority of planning work takes place in certain area of $Q$ some of the work is wasted. Increasing the sampling resolution leads to longer planning times. Alternate strategies are to sample close to obstacles to enable close-quarter movement without collision[CLH+05, p.216]. Another approach is to create a sparse graph by building on the concept of visibility by only adding new nodes that are occluded by obstacles from the vantage point of all other nodes, i.e. they »lie in the shadow« of obstacles.[CLH+05, p.218] Grid-based planners sample the space along the points of a grid, creating a very uniform distribution. Using hierarchical data structures the resolution can be increased spatially.

The strategy to connect adjacent nodes should also be selected with care. A standard approach is to connect each node to $k$ closest neighbours, which would lead to short connections but cluttered graphs. The opposite approach would be trying to create a sparse roadmap, where edges are only created if this increases the connectivity of the graph [CLH+05, p.225]. Using a lazy evaluation approach, checking for collisions only when necessary can lead to very efficient planning. The connections to the $k$ neighbours are assumed free of collisions. Once the query is started, these paths are tested on-the-fly.

Advanced sampling-based planners exist, most of them created specifically for one-shot planning. An example of such a planner would be Expansive-Space Trees (EST), that lends itself to kinodynamic planning too. In effect, the planner grows two trees, $T_{init}$ growing from the start point and $T_{goal}$, growing from the goal position. They grow towards each other until they can be merged into one. The advantage over PRM is that with this approach only the part of $Q_{free}$ is covered that is really needed for the query.

The algorithm employed in this thesis shares certain aspects with roadmaps, especially the concept of lazy evaluation.

# Chapter 3

# Integrating an arm into the Robbie software stack

## 3.1 Basics

### 3.1.1 General conventions and definitions

All file names are relative to the root of the Robbie software stack as found in the university subversion repository. The Robbie stack is not generally accessible to the public, so the sections discussing Robbie-specific changes might not be helpful to external readers. Also all Robbie components (Modules, Workers, Devices and Messages) are written in upper case to differentiate them from the concepts associated with these terms.

File names are written in `bold`, variable, class and method names in *italics*. Member variables begin with »m_«. All angles are stored internally as the data type double (with double precision). The term »configuration« is used as follows: A robot configuration of a robot system is »a complete specification of the position of every point of that system. The configuration space, or C-space, of the robot system is the space of all possible configurations of the system [...] The number of degrees of freedom of a robot system is the dimension of the configuration space, or the minimum number of parameters needed to specify the configuration.«[CLH+05, p. 40].


**Figure 3.1:** Axis enumeration

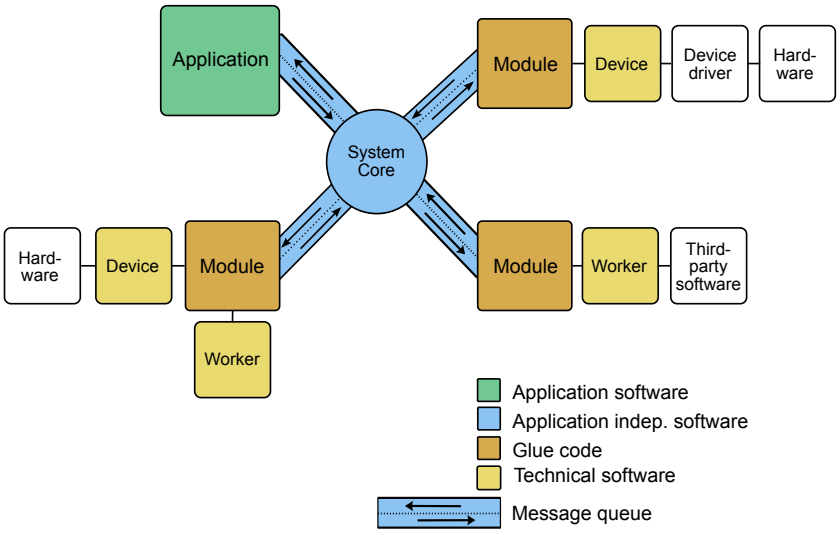


**Figure 3.2:** Schematic overview of the Robbie architecture

When referring to the anatomy of the Katana arm, the axes are enumerated for simplicity's sake, starting from the base and ascending to the gripper. Figure 3.1 depicts the enumeration. The following description assumes the viewer looks at the back of the arm, with the arm sitting on top of a table. The first or lowest axis is the rotational base embedded into the »foot« of the robot, which rotates around the y-axis. The second axis sits on top of the first and rotates around the x-axis. The third and fourth axes connect the first and second or respectively the second and third limb and rotate around the x-axis. The fifth axis rotates the gripper around the z-axis, while the sixth axis opens and closes the gripper.

### 3.1.2 Anatomy of the Robbie software stack

As mentioned briefly in section 1.2, the Robbie software stack is designed to be highly modular. The same code base should be usable for simulating a robot, making a Rescue league robot autonomously explore a maze, monitoring the Rescue mission over a network, playing back a sensor log file to test software changes in the lab without using a real robot and, of course, participating in the @Home league.

This requirement leads to five basic building blocks of stack components that interact in a well-defined manner. The system is modelled after the Mediator design pattern, whose intent is defined in [GHJV95, p. 305] as »Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently«. This is realized as a message passing system where

a central core will store subscription requests from components and allow each component to send messages to all other components that have subscribed to this kind of message. This system can be implemented efficiently in C++ through the use of pointers to messages. The architecture is shown schematically in figure 3.2.

Each participant of the message system is called a Module. A Module runs in its own thread of execution and subclasses *Module* or *ActiveMessageModule*. It can send messages at any time after initialization, which will be received by the core and distributed to other Modules. During initialization, the Module tells the core what message types it wants to subscribe to, and if it wants to receive only the latest message of each type, or each message. The Module has an inbox, into which incoming messages are sorted. On reception of a new message, the method *processMessages* is called, which can query the inbox for each message type. After a message has been acted upon, it can be flagged for deletion.

These messages are implemented as subclasses of *Message*, which lays the foundations for serialization. Serializability is an important aspect especially for sensor data, as these can be logged to disk and then deserialized at a later stage in log file playback. In the scope of this document, a subclass of *Message* will be termed (upper case) Message. Each Message has an associated type. Types are defined in a central registry and form the identifier for the subscription process. To facilitate archiving log files, Message instances include a version number, and the deserialization code must be able to unthaw older versions.

The next component is the Worker, a code block that can be re-used from different Modules. Generally all shared code is to be grouped into a Worker. The last component type is the Device, driver code for talking to hardware. Both Devices and Workers are concepts of the semantic level and do not subclass specific classes or implement certain interfaces. Conceptually, Workers can be instantiated often and used within any Module, Devices should only be instantiated once and used from within a Module that is specific for this Device, typically found in `Modules/Hardware`.

The system is configured via an XML config file. Here we define profiles to set variables that the stack can read at run time. Profiles can include other profiles and overwrite certain settings in the process, providing inheritance. So a logfile playback process will load the same configuration as the real game, but will additionally load the playback module. Also configurable via XML is the model of the robots physical geometry, the scenegraph. The Module *SceneGraphModule* always keeps an up-to-date version of the scenegraph, incorporating any changes like rotation of appendages or the pan-tilt-unit or movement of the platform. This scenegraph is broadcast periodically via a Message. Other Modules can load their private copy of the scenegraph at any time, which will then reflect the initial configuration of the robot, not the up-to-date one.

Integrating a robotic arm into the stack necessitated changes and additions to Modules, Devices, Workers and Messages.

## 3.2 Robot arm interface and hardware abstraction

Integrating the arm into the robot consists of several, loosely coupled tasks.

The most obvious of these is the installation of the actual hardware onto the robotic platform Lisa. In the course of development, this was shifted back as far as possible, so that work on the arm would not stall other activities that need the robot. Because the RoboCup team needed the arm mounted on the platform mid-way through the project, a virtual robotic arm needed to be implemented so that development was not bound by time constraints of sharing the robot between different projects.

So the first action was to write a hardware Device, virtual arm Device and hardware abstraction layer for the stack, as all other aspects depend on this. Then a control Module for robotic arms was implemented, followed by a graphical user interface and a path planning Module. The last action item was hooking the control Module into the sensor data and Message system and the integration of the arm into the @Home games.

It should be pointed out that certain restrictions exist for Robbie Devices in general and Human-Computer Interaction restrictions for RoboCup, both of which influenced the software design process. These shall be glossed over first.

### 3.2.1 Restrictions for Robbie Devices and Modules

Devices in the Robbie stack are run in the context of a Device-specific Module and hence in their own thread of execution. They typically have full access to the hardware and need to maintain little to no state. Their interaction with the rest of the stack is limited to whatever API they want to offer to their Module. Still, the Device code that sits between the stack and the hardware needs to fulfill certain requirements that the stack imposes on hardware drivers:

1. Non-blocking: Calls into Device code should not block unless absolutely necessary, so that the calling Module can fulfill periodic tasks.

2. Emergency stop:The hardware needs to be able to respond to the emergency stop Message if it is an actuator by stopping the motion of all movable parts, i. e. by exposing a pause method to the Module. This is assumed to be an important rule in robotics in general and also a rule for RoboCup, which will be enforced and tested by the jury. The movement should continue seamlessly once the un-pause command is called.

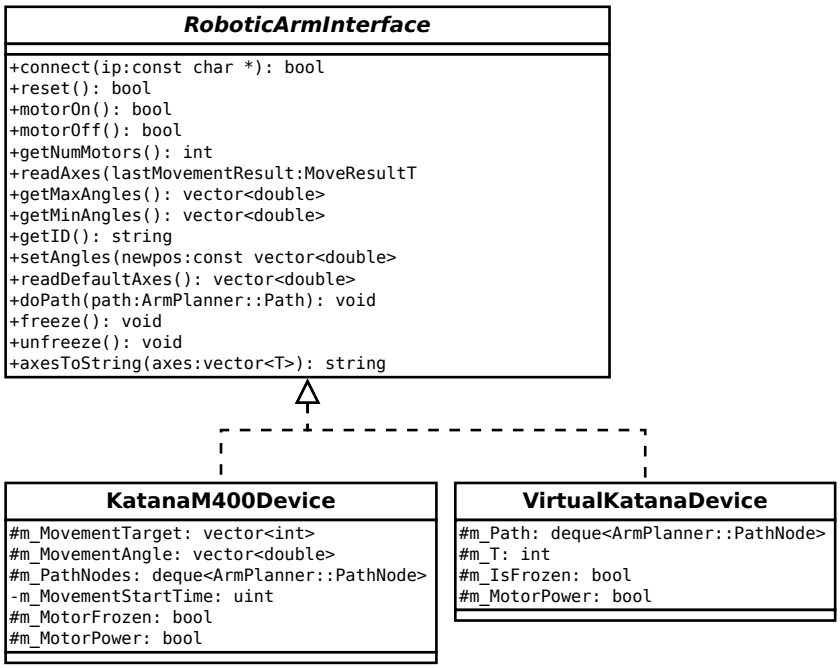


**Figure 3.3:** Class diagram for the hardware abstraction interface RoboticArmInterface and its implementations KatanaM400Device and VirtualKatanaDevice

3. Error robustness:The driver should not throw exceptions or leave the hardware in an unknown state and must recover gracefully from any error conditions.

4. Safety:The Device should move actuators in a way that cannot harm humans or damage the robot or environment.

These items will be referenced whenever corresponding code is examined.

To enable an easy transition between the virtual and physical arm, a programming interface `Devices/KatanaArm/RoboticArmInterface` for robotic arm drivers needed to be created. This is not a generic interface for all possible robotic arms, but only for arms that are similar to the Katana M400 arm, hence the folder name.

### 3.2.2 Abstract base class: RoboticArmInterface

The interface is implemented as an abstract virtual class and is the base class for `KatanaM400Device` and `VirtualKatanaDevice`, as shown in figure 3.3. The physical device driver shall be examined more closely, as the virtual driver merely mimics behaviour of the physical one.

### 3.2.3 Physical arm device: KatanaM400Device

The Device for the Katana 400 series uses the official driver from Neuronics, the »Katana Native Interface« [1] (KNI). For this implementation version 4.2.0 was used. KNI consists of a low-level wrapper for motor controller command submission, a high-level interface which accepts movement commands, and a kinematics library. Apart from the kinematics library the source code is available under the GNU Public license. An analysis of the offered solutions showed that the high-level library is a good foundation. The low-level code requires in-depth knowledge of motor controller commands and offers no advantage over the high-level interface.

In the *connect()* method, the Device connects to the robotic arm and reads the hardware revision. The Module will next call *reset()*, which will reset the hardware into a known state, read the number of axes and calibrate the motors if necessary. Calibration is required after the Katana was powered down. The process involves moving all motors to their mechanical stops and reading minimum and maximum encoder positions. The Katana will not execute move commands until it is calibrated. Unconditionally executing calibrations takes too much time if the robot application needs a restart during a competition, so the Device executes a very small movement of the gripper axis in the *reset()* method, and if this fails (KNI throws an exception when an uncalibrated motor receives a move command), calibrates the arm.

A very central method is the *setAngles* command, which tells the Device to move the arm into the given goal configuration. The position is passed as a vector of angles, one for each axis. If the goal configuration is not within the configuration space as established during arm calibration, false will be returned as error code. Due to restrictions »non-blocking« and »emergency stop« from chapter 3.2.1 the setAngles call cannot block. The non-blocking version of the corresponding KNI command *moveRobotToEnc* is called to start the movement. The goal configuration is stored in the member variable *m_MovementTarget* so we can check if the goal configuration has been reached.

Conversely, the *getAngles* method does not only read the configuration of the arm, but also checks if the last movement has finished. It is the ideal candidate for this check, as it is called periodically from its Module. If the last movement could not be completed, the referenced variable *lastMoveResult* will be set to »FAILED«. This typically happens when the goal configuration cannot be achieved by the arm although it is within the configuration space. Polling the movement state is a requirement for restriction »non-blocking«.

The Device also has the capability to execute complex movement operations called paths. A path consists of an ordered list of configurations. The associated

[1] Available from `http://www.neuronics.ch/cms_de/web/index.php?identifier=downloads`

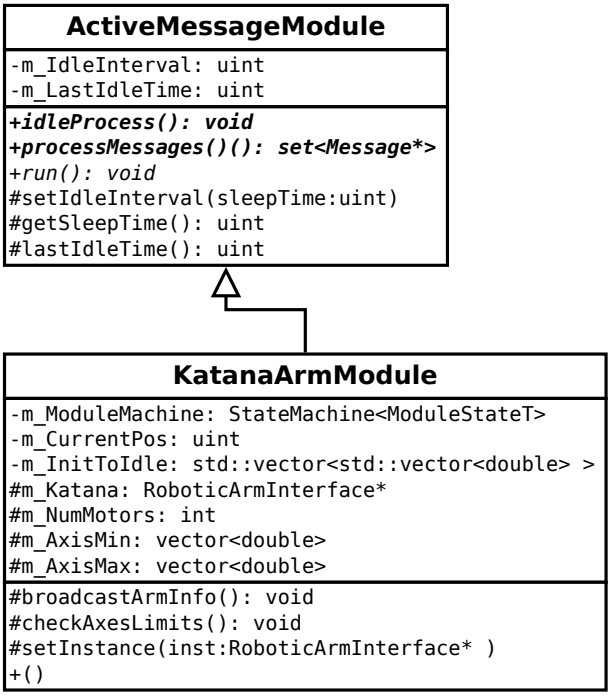


**Figure 3.4:** Class diagram for the hardware Module KatanaArmModule

method is *doPath*. The call does not block either, to observe restrictions »non-blocking« and »emergency stop«. Hence, the caller should regularly poll the state by calling *isMoving()*, which returns true if the arm is still moving.

### 3.2.4 Arm control Module: KatanaArmModule

A Device is always interfaced by a control module, in this case `KatanaArmModule`. A class diagram is shown in figure 3.4. Upon construction, it checks the central Robbie config file to see if the physical or virtual arm should be used for the running profile and instantiates the member *m_Katana* correspondingly. Also a state machine is initialized that tracks the arm state from disconnected over initialized to idle, and oscillates between this state, moving and, as a worst case, collided in case of a motor crash.

After initializing a Module, the core calls its *init()* method. Here the Device is probed, connected to and then reset. If no error occurred during reset, minimum and maximum angles are read. An arm information Message of type *RobotArmInfoM* is sent, containing the number of axes and minimum/maximum angles. The arm can also execute a set of initial movements to get into a known starting configuration. This process is started here if requested by the profile.

Next, the Module waits for incoming Messages and acts upon these. It subscribes to Messages of types *RobotArmMoveM* (sets a certain configuration for all axes), *RobotArmMoveAxisM* (changes the angle of one axis), *RobotArmPathM* (execute a movement path) and *PioneerDataM* (check for emergency stop). It basically exposes the Device API to the rest of the stack, doing sanity checking and error handling in the process.

As a so-called active Module, it will periodically wake up and execute the *idleProcess* method. Here, the configuration of the arm and its state is read and broadcast in a *RobotArmStateM*. If the arm was moving and has either succeeded in attaining the configuration or failed to do so, a *RobotArmMoveFinishedM* is sent with the status of the movement.

### 3.2.5 Arm path planning Module: ArmPlanModule

The ArmPlanModule is responsible for path planning and processing. It is not specific to robotic arms similar to the Katana M400, so the word is omitted in this class name. The algorithms used are the topic of chapter 3.4, the discussion here will focus on the interface.

The path planning Module is quite complex, as can be seen in the class diagram in figure 3.5. During construction it reads its configuration values and initializes a private copy of the scenegraph. A state machine is initialized, too. The Module then subscribes to the Messages *SceneGraphM* (copy of the up-to-date scenegraph), *RobotArmInfoM* (information on the robotic arm hardware like min/max angles), *RobotArmStateM* (configuration of the current robotic arm and movement state), *RobotArmPlanM* (path planning request), *PointCloudM* (three-dimensional sensor data for obstacle avoidance).

Before a *RobotArmInfoM* is received, no work can be done by the Module. After receiving such a Message, the contained minimum and maximum angles are used to pre-calculate a set of transformation matrices needed for the Forward Kinematics, one each for all possible rotational angles for the first four axes (counting from the base, so the gripper axes are ignored). The reception of a *RobotArmStateM* triggers the calculation of the Forward Kinematics, transforming the current arm configuration into the location of the end effector in working space, along with the distance to the closest obstacle. This data is sent in a *RobotArmPoseM* for visualization.

The real work of the planner starts after reception of a *RobotArmPlanM*, the path planning request. It contains a starting configuration and the goal effector position. Originally planned but not implemented was a way to tell the planner to add an item into the scenegraph, placing it in the gripper. This is needed when an actual item is held by the gripper, as otherwise the planner might propose a path that will cause this item to collide with obstacles. This item is always modelled as

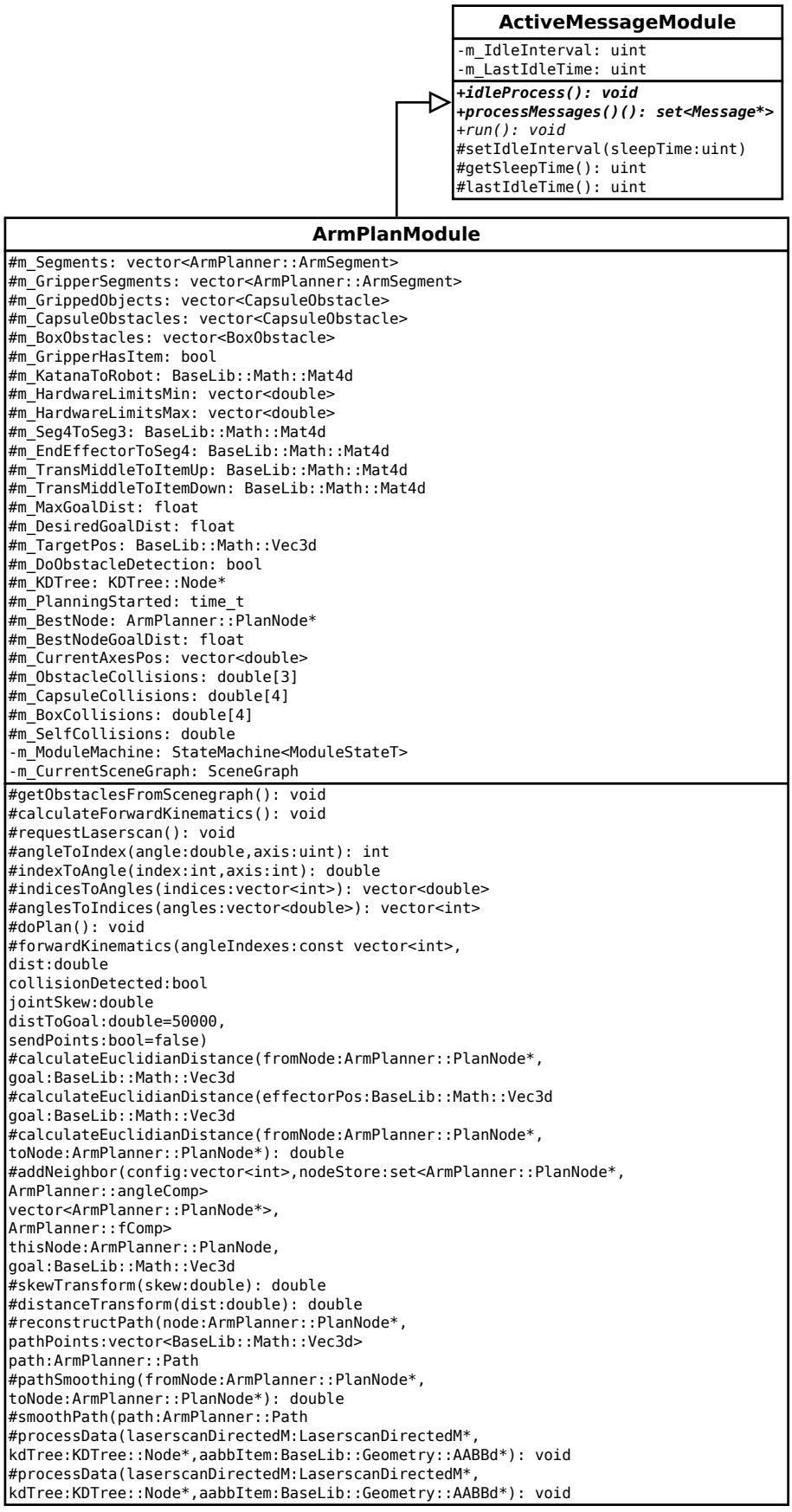


**Figure 3.5:** Class diagram for the path planning Module ArmPlanModule. Some member variables removed for legibility.

a capsule, a cylinder with half-spheres at both ends. The length of this bounding capsule can be specified in the Message. This is not implemented yet.

After this Message is received, the path is calculated as per the algorithms shown in chapter 3.4. The resulting path will be the shortest possible, but might be composed of many path points that only cause a slight change in the configuration of the arm. The path is smoothed to create a less complex copy that deviates only by a fixed distance from the original path, so that collision avoidance is not jeopardized. This path is then sent in a *RobotArmPathM*, which the *KatanaArmModule* receives and passes on to the Device for execution. The completion of the planning process is signalled to the software stack by broadcasting a *RobotArmPlanFinishedM* Message. If an error occurred during planning, the error condition is signaled in this Message too.

## 3.3 Foundation for path planning

Most path planners need utility functions for Forward Kinematics and collision avoidance. Before diving into the specifics of the chosen path planner, these generic functions and their implementations need to be investigated.

### 3.3.1 Forward kinematics and collision avoidance

Forward kinematics are used to calculate the pose of the robot from the configuration.

Robotic arms fall under the category of chain-linked segmented robots. The forward kinematics for these can be calculated using Denavit-Hartenberg parameters [HD64]. These lead to a matrix that can transform a parameter in configuration space into the working space. For this project the working space coordinate for each segment was needed to re-use this information during collision avoidance. Collision avoidance and forward kinematics are executed in the same code block.

To acquire coordinates for the start and end points of each segment, a transformation matrix is accumulated. The matrix is initialized with the transformation needed to transform the coordinate system of the first axis into the robot coordinate system. This consists of a rotation taken from the configuration for this axis, and a transformation along the length of the first segment:

$$M_{Segment1\ to\ ArmBase} = M_{Rotation\ \ Segment\ 1} * M_{Translation\ length\ \ Segment\ 1}$$

Using this transformation matrix, we can translate a point in the coordinate system of the first arm segment into the coordinate system of the arm base by multiplication. This in turn can be transformed into the robot coordinate system

by multiplying with the transformation matrix $M_{ArmBasetoRobot}$. Multiplying transformation matrices that transform from each segment into its predecessor segment, until we arrive at a formula that translates a position in the configuration space into one in the working space, resulting in the end effector position. All segment locations are more than a by-product, they are noted and re-used for collision avoidance purposes.

Because motion primitives are changes to an axis by a fixed increment, in our case by one, two or three degrees, it is possible to pre-calculate these transformation matrices for all possible axis rotations and for all axes as

$$M_{Segment\ m\ to\ Segmentm-1}[j] = M_{Rotation\ by\ j} * M_{Translation\ by\ length\ Segment\ m}$$

with $j = minAngle...maxAngle$. This optimization is used by [bco10] too. The precalculation is done as soon as the minimum and maximum angles are received from the **KatanaArmModule**. The robot geometry is stored centrally in the scenegraph to avoid redundancy. This information naturally contains the length of each arm segment. This is used as the transformation along the arm.

All obstacle avoidance algorithms used need to be highly optimized, as they are called many thousands of times in the course of planning. We differentiate three kinds of obstacles. Static obstacles are fixed to the robot and remain at a fixed position in the robot coordinate system. Dynamic obstacles are segments of the arm that could collide with other arm segments. Although attached to the robot, these change position dynamically, although this position is known with a high degree of accuracy. Arm segments are modelled as capsules[2] The biggest group of obstacles are external obstacles. These are not attached to the robot and are determined by on-robot sensors as a three-dimensional point cloud. As with all sensors, there is a certain amount of error through noise or calibration issues, so to be on the safe side each measured obstacle needs to be enlarged. These three kinds of obstacles can have different kinds of geometry. Static obstacles can be modelled as a capsule or an axis-aligned bounding box, dynamic obstacles have to be capsules and external objects come as points.

External obstacles are received as a **PointCloudM** Message from the stack's sensors. This Message contains an unordered list of three-dimensional measurement points in the robot coordinate system. They are entered into a k-d tree[Ben75] for high-speed lookups. A k-d tree is a k-dimensional binary tree which subdivides the working space along a split axis when adding points. Both children of each node lie on different sides of this splitting plane. The tree is sub-divided until the bounding box surrounding all points in all subnodes is sufficiently small, in

[2]A capsule is a cylinder of radius r with half-spheres of radius r at both ends. Visualize it as a straight sausage.

this case 10cm, at which point leafs are generated, which contain the actual data points. Points are only added to the tree if they lie within the arms working radius to conserve resources.

For performance reasons, only the last arm segment and the gripper elements are checked against obstacles. The first and second segments cannot physically reach obstacles before the last segment collides with them.

Ideally, obstacle detection will also give the minimum distance to the obstacle. This information is very valuable for the planner to ensure that the planned route is indeed safe, and to maximize the distance to obstacles. Checking arm segments against capsules is based on the segment-to-segment distance test by calculating the »Closest Point of Approach«[Sun10]. Given two lines

$$L1 : P(s) = P_0 + s(P_1 - P_0) = P_0 + su$$

and

$$L2 : Q(t) = Q_0 + t(Q_1 - Q_0) = Q_0 + tv$$

[Sun10] explain

»In any $n$-dimensional space, the two lines $L_1$ and $L_2$ are closest at unique points $P(s_c)$ and $Q(t_c)$ for which $w(s_c, t_c)$ attains its minimum length. Also, if $L_1$ and $L_2$ are not parallel, then the line segment $P(s_c)Q(t_c)$ joining the closest points is uniquely perpendicular to both lines at the same time. No other segment between $L_1$ and $L_2$ has this property. That is, the vector $w_c = w(s_c, t_c)$ is uniquely perpendicular to the line direction vectors $u$ and $v$, and this is equivalent to it satisfying the two equations: $u * w_c = 0$ and $v * wc = 0$.«

This can be transformed via

$$a = u * u, b = u * v, c = v * v, d = u * w_0, e = v * w_0$$

to

$$d(L_1, L_2) = |P(s_c) - Q(t_c)| = |(P_0 - Q_0) + \frac{(be - cd)u - (ae - bd) * v}{ac - b^2}|$$

which gives the minimum distance. If the distance is less then the combined segment radii, a collision has occured.

When both lines are parallel ($ac - b^2 = 0$) a fixed position on one line is chosen.

Testing axis-aligned bounding boxes against capsules is computationally expensive, especially if the closest point of approach is to be computed. Instead of capsules, simple line segments were used. To account for the capsule radius, the bounding boxes were inflated by the capsule radius, If the clipping process clips away the whole line, there is no intersection [3].

[3]Based on `http://www.gamedev.net/community/forums/topic.asp?topic\_id=433699&whichpage=1&#2882637`

The distance between external obstacles and arm segments is straightforward. The k-d tree is traversed by a recursive call to the function **recurseTreeLineDist**, which first calculates the distance for each point in this tree node to the line and saves it if is the smallest distance found yet. Then it decides to follow only the first, the second or both children of each node. Both children are followed if the capsules segment straddles the splitting plane or if the segment does not straddle the plane but the capsule radius means it would. Otherwise, only the child that is closer to the segment is followed. Following means in this context that the function is called for the specified child or children. As only the leaf nodes have data points in them, only a small percentage of all dynamic obstacle geometry needs to be inspected.

### 3.3.2 Choosing a planner

Many motion planners rely on inverse kinematics to establish a valid robot configuration for the given goal effector position, examples would be the aformentioned Wave-Front Planner or most uses of Roadmaps. The planner then connects starting configuration and goal configuration within the configuration space. This simplifies the planners, as even a linear interpolation between start and goal configuration will lead to a path with a continuos movement. A good approach to incorporate obstacle avoidance for high-dimensional robots are according to [CLH+05] roadmaps or the conversion of obstacle working space geometry into configuration space geometry, which is non-trivial.

I found relying on inverse kinematics to have drawbacks that limit their usefulness severely, the most obvious of which is the complexity of the algorithms involved. Although Neuronics supplies a complete Inverse Kinematics library with their API, the source code is not available. The process becomes a black box with a simple API that will only take a start configuration and goal position. This issue would not be critical if the process itself where not very complex. Converting a three-dimensional robot pose into a six-dimensional configuration leads is not a straight 1:1 mapping - for a given pose there can be multiple configurations. Avoiding obstacles means that not all goal configurations that Inverse Kinematics offer up are feasible. If the initial goal configuration offered by the Inverse Kinematic library is insufficient, there is no possibility of calculating other configurations except by offering other start configurations. This in turn might lead to highly suboptimal goal configurations being emitted.

Other drawbacks of using Inverse Kinematics are that as [bco10] mention, IK as a numerical approach can generate visually »awkward« paths in the sense of not being the path a human arm would take. It is also difficult to incorporate additional constraints into the goal configuration, like keeping a glass of water in the effector gripper balanced evenly or not planning close to joint limits.

Because of these issues an alternate approach was investigated.

### 3.3.3 About graph-based planning with motion primitives

The idea of using graphs as data structures in motion planning is not novel. Indeed most planning algorithms use graphs internally. Roadmaps tend to use graphs very intensely. Typically these graphs contain nodes that signify valid configurations and the edges show collision-free paths between these configurations. The approach used in this project differs from this usage significantly and should not be confused with the latter.

To escape the need to rely on inverse kinematics, other methods of obtaining a valid configuration for a given effector position are needed. The idea of using a graph built on simple motion primitives emerged and we found that other researchers where already working on similar approaches through the slides of the presentation of Benjamin Cohens summer project at Willow Garage, where he talked about using motion primitives to plan in cluttered environments [4]. Ben Cohen sent me a preliminary paper he was working on, where he detailed his efforts and results using this approach[bco10]. The results seemed good, so we decided on this route.

The general concept is to build a directed graph with the nodes being valid configurations and the edges representing a single, atomic configuration change called a motion primitive. Typically this would be a minimal change on one axis, although [bco10] uses primitives consisting of a change in two axes at the same time. For this project only simple primitives for the first four axes where used, as the gripper configuration is not part of planning here. This gives a total of eight motion primitives. The edges have a weight that describes how optimal this configuration is. This optimality can be based on different criteria, we chose to maximize the distance to the next obstacle.

Initially, the graph only contains the start configuration. From here all motion primitives are expanded and added to the graph as new nodes. Configurations that would intersect the arm with an obstacle are not added to the graph. Then the edge weights are calculated. This process will be repeated until we get close enough to the goal position or all configurations have been expanded.

### 3.3.4 On heuristic searching

The importance of heuristic search in robotics is highlighted in [bco10]: »Heuristic searches such as A* search [PEHR68] have often been used to find such trajectories. There are a number of reasons for the popularity of heuristic searches. First, most

[4] `http://www.scribd.com/doc/20233019/2009-09-Ben-Cohen-SBPL`

of them typically come with strong theoretical guarantees such as completeness and optimality or bounds on suboptimality (...). Second, there exist a number of anytime heuristic searches that find the best solution they can within the provided time for planning (...) . Third, there exist a number of incremental heuristic searches that can re- use previous search efforts to find new solutions much faster when previously unknown obstacles are discovered [16], [9]. Finally, treating a planning problem as finding a good quality path in a graph is advantageous because it allows one to incorporate complex cost functions, complex constraints and represent easily arbitrarily shaped obstacles with grid-like data structures (...).«

[bco10] continue by highlighting why heuristic searches have not yet been used for »high-DOF robotic manipulators«, as the Katana arm is: High-dimensional planning problems lead to a huge and complex graph, making even informed graph search infeasible. The authors suggest limiting all motion to a pre-defined set of motion primitives to limit graph growth: »...the majority of complex motion plans can be decomposed into a small set of basic (small) motion primitives.«

## 3.4 Algorithm implementation

The algorithm used for this project shares the basic idea with [bco10]. As the goal here is to minimize planning costs as opposed to planning in cluttered areas under adverse conditions, the implementation details differ. The differences will be denoted.

### 3.4.1 The A-Star heuristic search

[bco10] use the anytime search algorithm ARA*[LGT04] that can deliver suboptimal results at any time but will improve on them as time goes on. I have found that, if a solution for the planning problem in our uncluttered environment exists, it will be found before the time limit is reached. Using an anytime search would not be beneficial in these circumstances, so a standard A* search[PEHR68] was used.

The A* algorithm is an informed graph search algorithm. Instead of searching breadth-first or depth-first until the goal node is found, an informed search will choose the next node to expand by consulting a heuristic function for all candidates. Each edge has a cost function associated with it. Resulting paths sum up all edge costs within the path to obtain the path cost. The algorithm will find the path with the minimal path cost.

Each node is associated with the values $f$, $g$ and $h$. $g$ is the path cost of the optimal path from the start node to this node and $h$ is the estimate for the cost to the goal as determined by the heuristc. $f$ is $g + h$. For each node, all

successors are placed in the open set, where they are sorted by descending $f$, so typically a priority queue is used. Each iteration, the algorithm will remove the first item from the open set and check if it is the goal node. If not, all successors are expanded and added to the open set if they have not been visited yet. The algorithm maintains a closed set of visited nodes for this check.

In this concrete implementation, each node contains the configuration it represents, and also stores the end effector position and the motion primitive that was executed to initialize this node (so the path can be reconstructed bottom-up after completion). The nodes are created on-the-fly as knowledge of the complete structure is not necessary for the search to work.

All nodes are of type *ArmPlanner::PlanNode*. Once a node has been created, it is stored in the set *nodeStore*, which compares the nodes on the basis of their configuration. If a node is to be in the open set, it is also added to the priority queue *openQueue*, which orders the nodes by their *m_F* ($f$) lowest-first. Nodes also have an attribute *m_IsOpen* to show if they are in the open set. All nodes in *nodeStore* which aren't in the open set are automatically in the closed set.

### 3.4.2 Graph construction

The graph initially contains one node representing the start configuration. This node is placed into the open set. The timestamp is stored in *m_PlanningStarted* to check for timeouts.

The algorithm then begins to iterate over the open set. The topmost item (the item with the lowest $f$) is popped off the open set and stored in *currentNode*. The distance from currentNode to the goal position is calculated. If the distance is the closest encountered yet, this node is saved in the attribute *m_BestNodeGoal* as the best node seen. If the distance is less than the lower limit *m_DesiredGoalDist*, then the goal has been reached. If the timeout has occured and the distance from the best node to the goal is less than the upper limit *m_MaxGoalDist*, the goal is considered as reached too, and *m_BestNodeGoal* will be used as *currentNode*. Once the goal is reached, the path is reconstructed and smoothed as described in 3.4.3 and then broadcast via an *ArmPathM* Message.

If the goal has not been reached, the *currentNode* is added to the closed set. Then new nodes are expanded, one for each motion primitive. Here, we add eight new nodes, as we increase and decrease the rotation of each of the first four axes by the delta value *stepSize*. This value depends on the distance to the goal. The closer the arm gets to the goal, the smaller the size of these changes. Good results where achieved with a step size of 3 degrees of change if the distance to the goal is larger than 10 centimeters, 2 if it is larger than 5 centimeters and 1 otherwise. If the resulting configuration of the new node is invalid (if it is not between minimum and maximum angles), it is discarded.

| Distance to next obstacle | Path cost multiplier |
|---|---|
| $> 90mm$ | 1.0 |
| $> 83mm$ | 1.2 |
| $> 77mm$ | 1.6 |
| $> 70mm$ | 2.0 |
| $<= 70mm$ | 2.2 |

**Table 3.1:** Path cost depending on distance to closest obstacle

All freshly expanded nodes ($v_{new}$) are then checked against the *nodeStore* to see if they exist already. Any that exist but are in the closed set are discarded. If a node exists but is in the open set, the path cost of the current path is compared to the cost of the older node ($v_{old}$). If the newer path is better i.e. if $g_{new} < g_{old}$ then the existing node is updated to reflect the new path cost and its predecessor node is set to *currentNode*.

If the node cannot be found in the node store, it's end effector position is calculated along with its distance to obstacles. This calculation has not been done at earlier stages of the algorithm to not waste this effort on nodes that would have been discarded anyway. If the node intersects an obstacle, it is directly added to the closed set. Otherwise $h$, $g$ and $f$ are calculated, and the node is stored in the node store and in the open set.

The path cost for each graph edge is derived from up to two parameters. The most important is the distance to the next obstacle. This obviously should be maximized, hence small values lead to big path costs. The current implementation is a simple distinction based on table 3.1.

The second parameter penalizes changes in arm velocity. If a node executes a certain movement primitive in the configuration space, successor nodes executing different primitives have higher path costs. A successor primitive that only causes slight motion deviation would incur less cost. The goal here is to create a path that is as smooth as possible. The idea was pioneered by [bco10]. In this implementation it proved counter-productive as explained in section 3.2.5 and currently always returns 1.0 as cost factor, thereby not changing path costs.

### 3.4.3 Path reconstruction and post-processing

Starting from the goal node found during searching, the standard A* recursive algorithm is applied to reconstruct the graph by calling *reconstructPath* with this node as argument. The motion primitive of this node is taken along with its end effector position to create a new *ArmPlanner::PlanNode* instance, which is then pushed onto the stack *pathPoints*. Then *reconstructPath* recursively calls itself with the predecessor node as argument. If the node has no predecessor, the

```
m = number of path nodes
p = path nodes from 0 to m
startidx = 0
limit = 10.0
smoothedpath = {p[startidx]}
while startidx < m-2:
        endidx = startidx + 2
        do:
                mididx = startidx + 1
                while mididx < endidx:
                        if distbetween p[mididx] and
                        line between p[startidx] and p[endidx] > limit:
                                add p[endidx-1] to smoothedpath
                                startidx = endidx - 1
                                mididx = endix
                        else:
                                mididx = mididx + 1
                endidx = endidx + 1
        while endidx < m
add p[endidx] to smoothedpath
```

**Figure 3.6:** The algorithm used for smoothing in pseudo code.

method returns, ending the recursion. The stack *pathPoints* now holds all path points in the correct order.

This path is optimal with respect to the cost function thanks to the properties of the A* search algorithm and the use of a valid heuristic. This can lead to inefficient paths, as skirting obstacles leads to jagged edges and »spikes« in the path. The limited number of motion primitives means that smooth circling motions are difficult to achieve. Therefore, the path is smoothed before being executed. Path smoothing is used in [bco10] too, although the algorithms used differ.

The idea behind the smoothing algorithm is to create a new path that may not deviate from the original path by more than a maximum distance (in this case 1cm) by dropping path nodes. The algorithm is shown in figure 3.6.

The smoothed path is broadcast via a *ArmPathM* Message and then executed the *KatanaArmModule*.

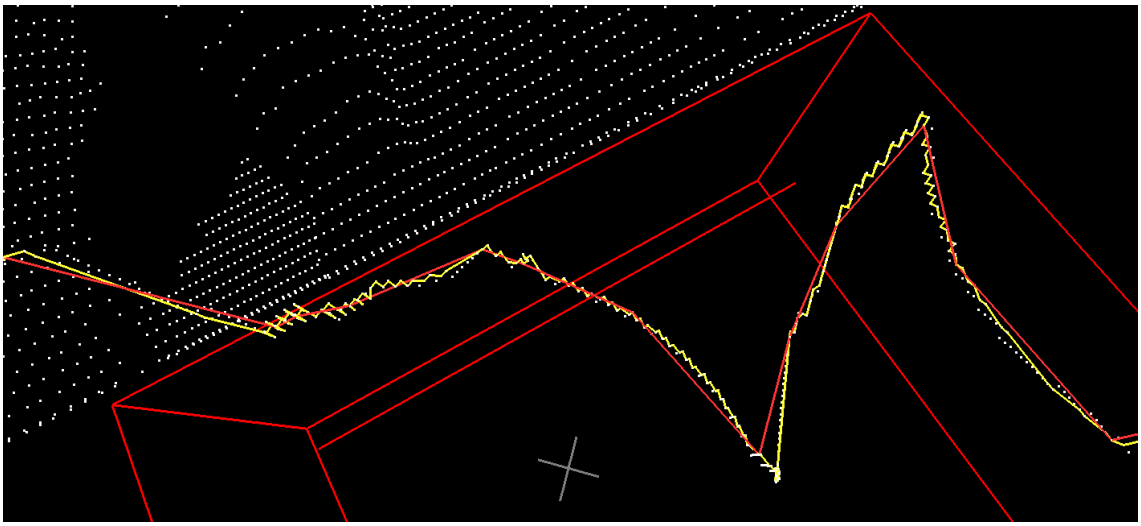

**Figure 3.7:** Path smoothing: The yellow line is the unsmoothed path, the red line close to it is the smoothed copy.

# Chapter 4

# Evaluation and lessons learned

As the stated goal of this thesis was to participate successfully in the Robocup @home league in manipulation-based games, this participation will be a cornerstone of the evaluation. Before robot and team travelled to Singapore, extensive tests and evaluations where performed, the results of which shall be the focus here.

## 4.1 Evaluating performance and success

Before getting into a specific setting, the overall performance and stable functionality of all Devices and Modules where evaluated. Considering that higher and complex functions of the stack require the successful operation of basic functions, the simpler operations need to be very robust. Also, Robocup security regulations for the safe interaction of robots and humans need to be observed, as failing these can be harmful or result in being banned from the competition. These regulations where outlined in chapter 3.2.1.

### 4.1.1 Basic functionality

The Device *KatanaM400Device* itself needs to either successfully perform a movement or path, or return the error state to the stack. We found that the Katana arm will not perform movement close to joint limits reliably. How close to each joint limit the motors will operate cannot be precisely measured as the effect is erratic and different on each joint. This was further complicated by the fact that moving too close to a limit resulted in a motor timeout, as the motor would just not execute the move command at all, but also would not reject it out of hand. The *ArmPlanModule* now adds a »dead zone« around all reported joint limits, a setting that is configurable via the parameter *fAxisDeadZone* in the configura-

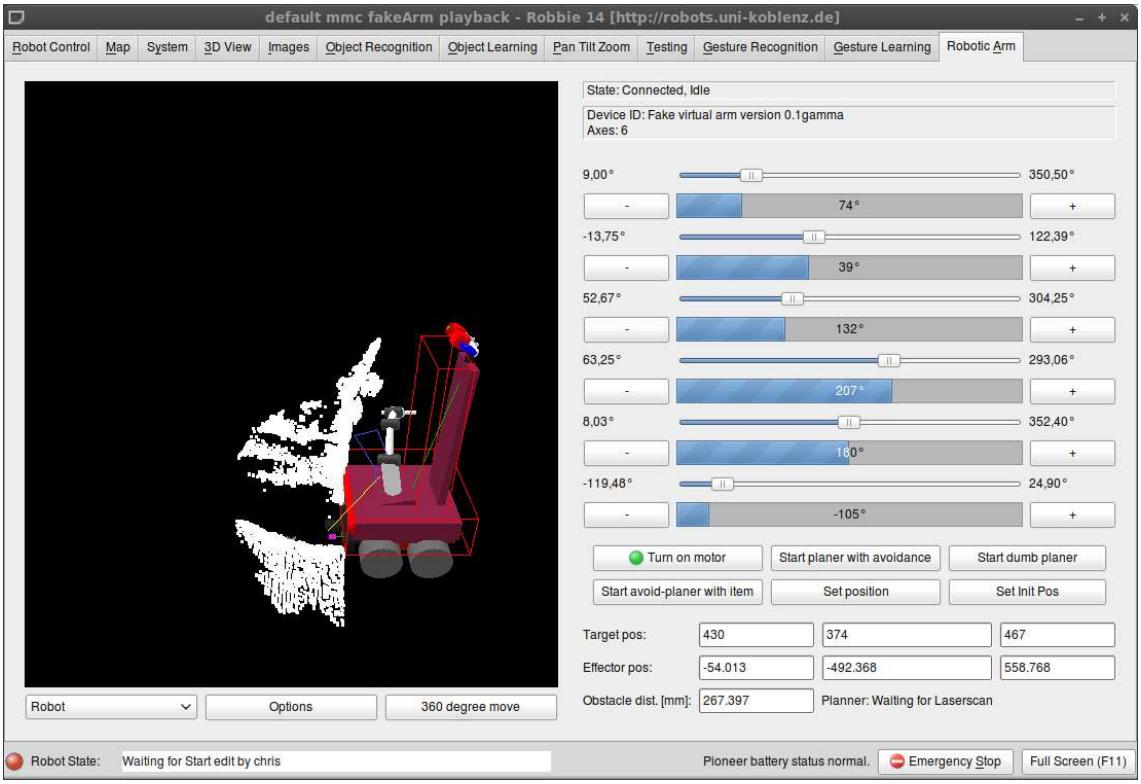


**Figure 4.1:** Screenshot of debug visualization. The blue wireframe arm on the left represents the configuration the planner is examining.

tion, the default value being eight degrees. After this alteration, move and path commands executed fine.

Hitting the emergency stop button on the robot should stop all arm movement as soon as possible. Initially, the **KatanaM400Device** used blocking calls when calling into the KNI library for move commands. This was changed to use non-blocking calls. On receiving the emergency stop signal, the KNI motor freeze command is issued by the Device. Motion is unfortunately not ceased immediately, it can take several seconds to stop. In most cases, the unfreeze command will also cause the arm to resume motion as planned. This goal was not reached completely and further work needs to be done in this area.

In order to reduce the time it takes to start an @home game, arm encoder calibration is only executed when necessary. This saves about 30 seconds of setup time.

### 4.1.2 Planning time and success rate

During early stages of development, the planning process would often time out, giving no result. To get a better understanding of the process involved, the planner will send status information along to the stack GUI, which will be displayed in 3D as shown in figure 4.1. This user interface not only shows the configuration that the planner is testing at this instant, but also allows manual movement of the arm, planning to a certain position and reading end effector position and the distance to the next obstacle. The OpenGL-based sensor status display proved a great tool to understand the workings of the algorithm and how to optimize it.

An in-depth analysis of the planning process was now possible. To measure planning performance, only the planner runtime was profiled. Benchmarks measuring total planning time produced varying results that could not be reproduced, as the complete Robbie stack needs to be running in the background, consuming CPU time. With a profiler, non-planner method runtime could be discounted, and the results proved stable. All measurements where taken on a machine with Intel Core 2 Duo 2.4 GHz under Ubuntu Linux 9.10. A complex scenario was used for planning: the arm had to evade three large obstacles placed on a table. This scenario and a valid path is shown in figure 4.2. The log file used is to be found on the accompanying CD and is called »Zwei_Hindernisse_links.log«. A path is planned from the starting configuration (74, 39, 132, 207, 180, -105) to the effector position of 430, 374, 467.

After ensuring correct algorithm execution, the code was profiled to identify »hot spots«. First, the collision avoidance was moved to the latest possible point in time, after having eliminated duplicates and invalid configurations. This lazy evaluation cut time spent on collision avoidance by 50%. Implementing the time-memory tradeoff of pre-calculating the transformation matrices for the first four arm axes as detailed in section 3.3.1 decreased forwards kinematics runtime by about 20%.

The next increase in planning speed was accomplished by removing all unnecessary square root calculations during distance functions. All distances are now expressed internally as squared distances in millimeters. The only time this is reduced to linear distance by performing a square root is when estimating the distance between the arm and an obstacle. This eliminated two calls to *sqrt* per loop and resulted in a decrease of collision avoidance and goal distance cpu usage of about 10%. Finally, collision avoidance was aborted as soon as possible, which further reduced time spent on collision avoidance by 10 to 20 %, depending on the length of the path.

A big decrease in planning time was achieved by decreasing the amount of obstacle sensor data points. Only points that are in the operational radius of the arm are considered now. Initially, another experiment was to limit the density

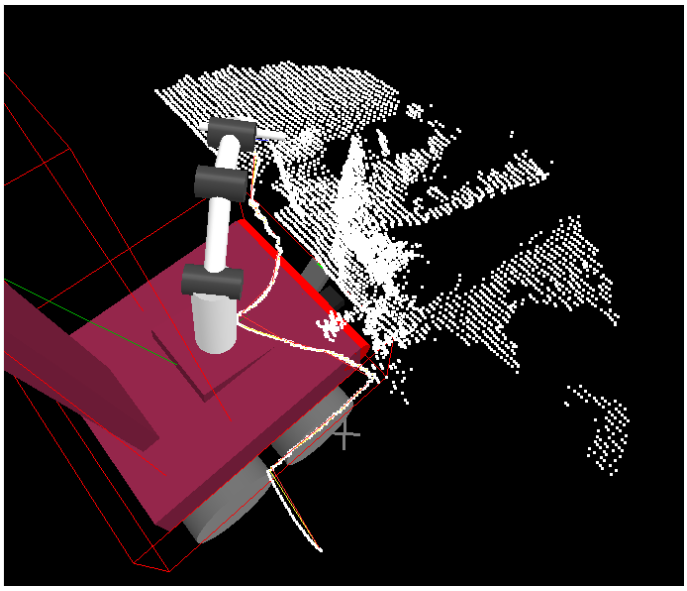

**Figure 4.2:** Screenshot of path planner having completed the complex path scenario used for evaluation

of sensor data points. Some points are positioned closely together. Avoiding one in collision avoidance would automatically avoid the other, so the additional information on obstacle geometry was not helpful but increased the number of points that the Module needs to test against. A minimum distance between sensor data points was added, but pre-processing data points took longer than an average path planning process, so it was removed for now.

The biggest gain was the introduction of a step size. In the initial version, the planner always changed an axis by the same fixed small amount when applying a motion primitive while expanding a new node. When the effector was still far away from the goal, this high resolution was not necessary. A variable step size was introduced as explained in section 3.4.2. Time spent planning dropped by about half. The risk of colliding with obstacles does not increase because obstacles are skirted by at least 5cm as enforce by the obstacle transform.

In the final version, the planner could examine between 1000 and 3000 nodes per second. In the complex example shown in figure 4.2, a path would be found in about two seconds. Only if additional constraints where added would timeouts still occur. Examples of these constraints are keeping a glass of water that is held by the gripper stable, or planning through very narrow gaps between obstacles.

Although most optimization work went into the collision avoidance complex, these operations still take the biggest chunk of CPU time. Figure 4.3 shows a treemap, a hierarchical distribution of CPU usage of the **doPlan()** method and

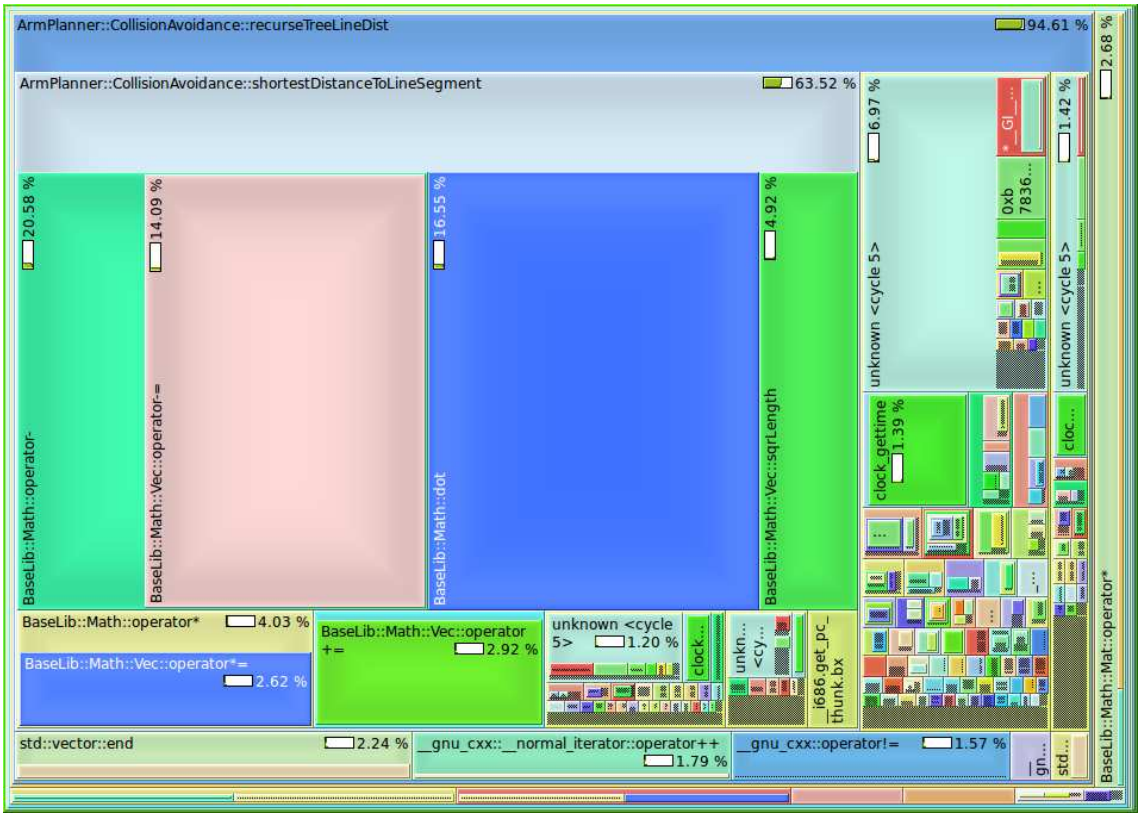


**Figure 4.3:** Treemap of path planning CPU usage distribution, acquired with Google Performance Tools and kcachegrind

all callees and child methods. All numbers are percentage of CPU usage relative to **doPlan()** itself. Only 5% of CPU time is spent other tasks apart from collision avoidance.

## 4.2 Evaluation of success in the @home setting

The team Homer@UniKoblenz participated at RoboCup 2010 in Singapore in the @Home league. The capability to grab objects from shelves or tables and, via the platform's built-in gripper from the floor enabled us to participate in all games that required manipulation skills.

The first test to use the manipulator was the »Robot Inspection Poster Session«. The idea of the game is for the robot to introduce itself to the jury and the spectators. The arm moved along a pre-defined path without doing dynamic obstacle avoidance. The jury was nevertheless impressed by the idea and the versatility of our manipulation code.

The next game where the team could show the manipulator was the Open Challenge. This game is very important because it has a high influence on the teams ranking. There is no fixed script, the robots are supposed to show their most advanced features while a team member does a presentation on the techniques involved. The team homer used their second robot »Waylon«, which followed Lisa around, carrying a litter box. Lisa was supposed to find trash on the stage floor, pick it up and place it into Waylons box. Although the gripping and dropping actions performed well, Lisa touched Waylon slightly at one point. The jury was impressed by the autonomous cooperation between both robots and the arm path planning collision avoidance algorithms involved (bearing in mind that some other teams use manually generated collision avoidance information). Team homer was now ranked third out of twenty-four teams.

The team made the fourth place out of a total of all 24 teams. Homer was awarded the »Innovation Award« for multi-robot coordination, good collision avoidance during manipulation and intuitive gesture control. During the competition the arm grabbed four obstacles with full obstacle avoidance, all of them successful.

All in all, the manipulation capabilities of the platform gave the team an advantage. Gripping items and placing them in shelves is a complex task that nevertheless can be performed by a few @home teams. Our advanced collision avoidance features proved ahead of the race and can be seen as a success. The goal of this project is accomplished.

# Chapter 5

# Outlook

The goal of gripping objects, placing them on tables or shelves and avoiding complex obstacles in real-time was accomplished by this project. Of course, various improvements are possible.

## 5.1 Further work

There are many parameters in the algorithm that can be tuned to increase planning speed or enable planning while observing a wide range of constraints. Most of these »knobs« are already implemented in the code and only require tuning of variables or the configuration file.

The *stepSize* parameter in the *doPlan()* method sets the angular movement increment (the »stride«) for each expansion of a new node. Increasing this parameter linearly increases planning speed. At the moment an adaptive approach is used that will decrease the step size as the distance to the goal decreases. Apart from increasing step size generally or allowing larger steps closer to the goal, the step size could be attuned to the distance to the next obstacle. This way, planning close to obstacles or through difficult patches could be optimized.

Path smoothing as part of the path cost function was introduced by [bco10], where it proved beneficial in cluttered environments. In this project, it was implemented but disabled because it did not show benefits to path smoothness in our more orderly environments. Further testing is in order to show how this concept can aid our planning approach.

The distance to the closest obstacle is an important factor in the path cost function. At the moment this is a non-linear function based on simple range-based function. A continuous function would probably result in a smoother path with less abrupt transitions. Likewise, the skew transform that is used to keep the arm tip stable should be based on a linear function. This transform could also use

more flexibility, like switching between a more constrictive skew transform when holding an object and more lax restrictions when speed is paramount.

More demanding sub-projects would be to revisit the collision avoidance approach. The majority of planning run time is spent on collision avoidance. Further limiting input data from sensors or utilizing alternate data structures like spatially clustering trees might dramatically increase the planner's speed. A first experiment here done by only adding new nodes to the k-d tree if they was a minimum distance to any neighbor node. Unfortunately the time taken for this check was longer than the runtime of the whole planning process. A more intelligent approach might yield good results here, especially using the tree itself to detect the closest neighbour.

The biggest single speed improvement might result from adjusting how often collision detection is performed. If the predecessor node showed that obstacles are far away, the successor node might skip collision avoidance completely and only do forward kinematics.

All optimizations mentioned are not required for successful path planning in the current setup. Adding more constraints to the path cost function can dramatically increase planning time and might render these optimizations necessary. Figure 4.3 in section 4.1.2 should offer starting points for in-depth optimization and other reworking.

Alternatively, the A* search might be replaced by an anytime algorithm like ARA*[LGT04]. Using this algorithm, even tough constraints could be applied, and at the same time it would be possible to see after only a few seconds of planning if a solution exists at all.

The emergency stop solution used at the moment is insufficient. The arm might not stop immediately and will sometime not resume the movement after the emergency stop is removed. Further work is needed here, although this might be a limitation of the arms firmware.

A very interesting project will also be to implement collision avoidance with objects residing in the gripper.

## 5.2 Acknowledgments

This work was made possible by the Active Vision Working Group (AGAS) at the University of Koblenz-Landau headed by Prof. Dietrich Paulus. Many thanks to the AGAS team and to the members of the RoboCup team for their assistance in this project.

Special thanks to David Gossow for his work on optimizing this framework onsite at Robocup Singapore and the IJCA in Anchorage, Alaska and to Sebastian Vetter for supervising this thesis together with David. I am also indebted to Frank

Neuhaus who came up with the original idea for the heuristic search with motion primitives as path planner. This work was also greatly helped by the cooperation of Benjamin Cohen et al, wo gave me access to a review copy of their paper [bco10] that they where preparing for IJCA.

# Bibliography


[bco10] *Search-Based Planning for Manipulation with Motion Primitives*. Anchorage, Alaska, 2010

[Ben75] Bentley, Jon L.: Multidimensional binary search trees used for associative searching. In: *Commun. ACM* 18 (1975), Nr. 9, S. 509–517. `http://dx.doi.org/http://doi.acm.org/10.1145/361002.361007`. – DOI http://doi.acm.org/10.1145/361002.361007. – ISSN 0001–0782

[CLH+05] Choset, Howie ; Lynch, Kevin M. ; Hutchinson, Seth ; Kantor, George A. ; Burgard, Wolfram ; Kavraki, Lydia E. ; Thrun, Sebastian: *Principles of Robot Motion: Theory, Algorithms, and Implementations*. Cambridge, MA : MIT Press, 2005

[GHJV95] Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Design patterns: elements of reusable object-oriented software*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. – ISBN 0–201–63361–2

[GWB+10] Gossow, David ; Wojke, Nicolai ; Bing, René ; Buchholz, Urs ; Schrage, Robin ; Mützel, Andreas ; Read, Kevin ; Thierfelder, Susanne ; Vetter, Sebastian ; Paulus, Dietrich: RoboCup 2010 - homer@UniKoblenz (Germany) / Universität Koblenz-Landau. 2010. – Forschungsbericht

[HD64] Hartenberg, Richard S. ; Denavit, Jacques: *Kinematic Synthesis of Linkages*. New York: McGraw-Hill, 1964

[LGT04] Likhachev, Maxim ; Gordon, Geoff ; Thrun, Sebastian: ARA*: Anytime A* with Provable Bounds on Sub-Optimality. In: *IN ADVANCES IN NEURAL INFORMATION PROCESSING SYSTEMS 16: PROCEEDINGS OF THE 2003 CONFERENCE (NIPS-03*, MIT Press, 2004

[PEHR68] P. E. Hart, N. J. N. ; Raphael, B.: A formal basis for the heuristic determination of minimum cost path. In: *IEEE Transactions on Systems, Science, and Cybernetics, SSC-4(2):100?107* (1968)

[Sun10] Sunday, Dan: *http://softsurfer.com/Archive/algorithm_0106/algorithm_0106.htm* 2010