University of Stuttgart
Institute of Industrial Automation and Software Engineering
Prof. Dr.-Ing. Dr. h. c. P. Göhner
University of Western Australia
School of Electrical, Electronic and Computer Engineering
Prof. Dr. rer. nat. habil. T. Bräunl

Diplomarbeit No. 1882

Swarm Clustering System with Local Image Processing and Communication

Jia L. Du

Engineering Cybernetics
Student # 1850152

Aug 2002 – Jan 2003

Abstract

In this thesis a collective clustering algorithm is presented. The used robots are behavior-based and fully autonomous. The developed multi-robot system clusters randomly distributed cubes in a walled area. A framework for behavior-based control was developed. Four behaviors were implemented: exploration, avoidance, pushing, and communication. Inter-robot communication is used to speed up the emergence of a common cluster point. On-board image processing is used for the detection of the cubes. The clustering algorithm was implemented both for real robots and a simulator. Using the developed software the robot colony successfully completed the task. And the colony proved to be scalable and robust.

Kurzfassung (German)

In dieser Diplomarbeit wird ein Clustering-Algorithmus für ein Multi-Roboter-System präsentiert. Die einzelnen Roboter arbeiten völlig autonom und verhaltensbasiert. Das entwickelte System trägt in einem von Wänden umschlossenen Gebiet zufällig verteilte Würfel an einem Punkt zusammen. Hierzu wurde ein Framework für eine verhaltensbasierte Robotersteuerung entwickelt. Vier Verhalten wurden implementiert: Erkundung, Ausweichen, Transport, und Kommunikation. Kommunikation zwischen den Robotern wird verwendet, um das Finden eines gemeinsamen Clusters zu beschleunigen. Lokale, also robotereigene Bildverarbeitung wird für die Erkennung der Würfel genutzt. Der Algorithmus wurde sowohl für reale Roboter als auch für ein Simulationssystem implementiert. Die mit der entwickelten Software arbeitende Roboter-Kolonie erfüllt ihre Aufgabe erfolgreich, und erweist sich als skalierbar und robust.

Acknowledgements

First, I would like to thank my supervisor Prof. Thomas Bräunl. Without his support and advice, and the invitation to the University of Western Australia, this thesis would not have come into existence.

I also would like to say many thanks to Prof. Peter Göhner for supervising the project from my home university. Only his support made the this project possible.

Furthermore, I would like to thank Jens Konnertz for his support and supervision of the software engineering part of this project.

Burkhard Iske made a major contribution to this thesis by providing the source code for his Khepera cube clustering software. Although the finally developed software differs considerably from his program, the studying of his code gave me the knowledge and the confidence to design and implement my own clustering algorithm.

Parts of the image processing algorithms of this project are based on the experiences that Birgit Graf made with her soccer program for EyeBots. Thanks!

Many thanks to Axel Waggershauser for his support with the EyeSim simulator.

I would also like to thank the UWA staff, especially Sandra and Brenda, for their assistance during my stay at UWA.

Last, but not least I would like to thank Antonio, Christoph, Jochen, and Sid for their support during the creation of this thesis, and for all the fun we have had working together.

Table of Contents

Figure Index

Index of Code Examples

Abbreviations

IAS	Institute of Industrial Automation and Software Engineering
CIIPS	Centre for Intelligent Information Processing Systems
UWA	University of Western Australia
PSD	Position Sensitive Detector
SIR	Sensors and Image Recognition
RoBIOS	Robot Basic Input Output System

1. Introduction

The use of simple and reactive behavior-based robots has become increasingly popular among researchers in robotics in recent years. Behavior-based robots are characterized by a tight coupling of sensing and action, and the absence of world models. This results in simple, robust and reactive individuals, as opposed to model-based robot control systems, which are complex, little robust, and hard to maintain [13].

Rapid advances in computer technology and robot hardware components have also had a major impact on robotics research. Firstly, the development and usage of large scale multi-robot systems has become feasible with the decrease in component costs. Multi-robot systems are expected to have significant advantages over single-robot systems including an increased performance, the ability to solve more complex tasks, and an increased robustness due to redundancy and simpler individual robots. Secondly, the enormous increase in processing power on the one hand, and the miniaturization of hardware components on the other hand, have made the development of small, fully autonomous robots with on-board image processing possible.

The term *swarm intelligence* describes the approach to use behavior-based robots to create an autonomous, robust, flexible, and scalable multi-robot system. It is inspired by biological systems that consist of many agents, for example ant colonies. Typical tasks for such swarms include exploration and transportation.

The objective of this project is to develop a swarm clustering algorithm. The robot colony, consisting of fully autonomous and behavior-based agents, is to cluster randomly distributed cubes in a walled area. The swarm should be robust and scalable. Some form of robot communication is to be incorporated to ensure that a single cluster point remains in the end. And on-board image processing should be used for object detection. The goal is to demonstrate the feasibility, scalability and robustness of a swarm clustering system with on-board image processing and communication.

The thesis is a joint project of the Institute of Industrial Automation and Software Engineering (IAS), University of Stuttgart, Germany, and the Center of Intelligent Information Processing Systems (CIIPS), School of Electric, Electronic and Computer Engineering, University of Western Australia, Perth. The work is carried out at the Mobile Robot Lab at CIIPS.

The project duration is 6 months. See the project plan for a detailed project structure plan, milestones and bar charts.

Chapter 1 is this introduction. In chapter 2, I give an overview of the necessary basics. In chapter 3, the requirements to the software to be develop are stated. The system model is presented in chapter 4, and the system architecture in chapter 5. In chapter 6, I will go into the implementation details. Chapter 7 quickly summarizes the performed system tests. In Chapter 8, I give an evaluation of the developed system and suggestion for future work. In chapter 9, I conclude the thesis.

2. Methods and Materials

2.1 The EyeBot

The EyeBot is a fully autonomous mobile robot. It does not rely on sensor data from external devices or remote processing resources and power supplies.

Each EyeBot is equipped with a 32 -bit controller at 33MHz, and up to 2048 KB RAM. The processing capacity of the controller is sufficient for most image processing tasks. To give an idea, in this project the computing power was sufficient to handle one to two frames per second, in addition to other computations for motion control and user interface.

The robot has a differential steering system, with 2 DC motors with shaft encoders for the left and right front wheel. The rear part is simply dragged along. The differential steering system allows the robot to drive forward and backward, to turn on the spot, and to drive curves, depending on the level of activation of the left and right motor. Dead reckoning is used for position estimation by counting the encoder ticks. However, position estimation using dead reckoning is inherently inaccurate - the error accumulates over time. A magnetic compass allows the correction of the orientation.

The main sensor input is provided by a digital color camera. It takes 24 -bit RGB images at a resolution of 60*80 pixels. Furthermore, the robot has three position sensitive detectors (at the front, at the front to the left, and at the front to the right). They determine the distance to an obstacle by measuring the time till the reflection of a previously sent out infra-red signal is registered. The estimated distances are excellent for distances between 10 cm and 20 cm. However, for very small obstacle distances the sensor readings tend to be too large. The signal bounces more than once until it is registered by the receiver [1].

Each robot has a wireless communication module that can be used to communicate with other robots. A network is automatically established, it operates as a virtual token ring and has fault tolerant aspects. A net Master is negotiated autonomously, new EyeBots are automatically integrated into the net, and dropped out EyeBots are eliminated from the network.

Finally, for a direct communication with the user, each robot has a graphics LCD and four input buttons.

The EyeBots can be programmed in C/C++ or assembler. The so-called RoBIOS operating system provides C functions to control the robot and to access the connected devices. In addition, a framework for multi-threading and basic image processing algorithms are provided. Programs for the EyeBot are compiled on a PC with a modified version of the GNU C/C++ compiler. Then the program can be downloaded to the robot using a serial connection. A list of the used RoBIOS functions can be found in the system architecture.

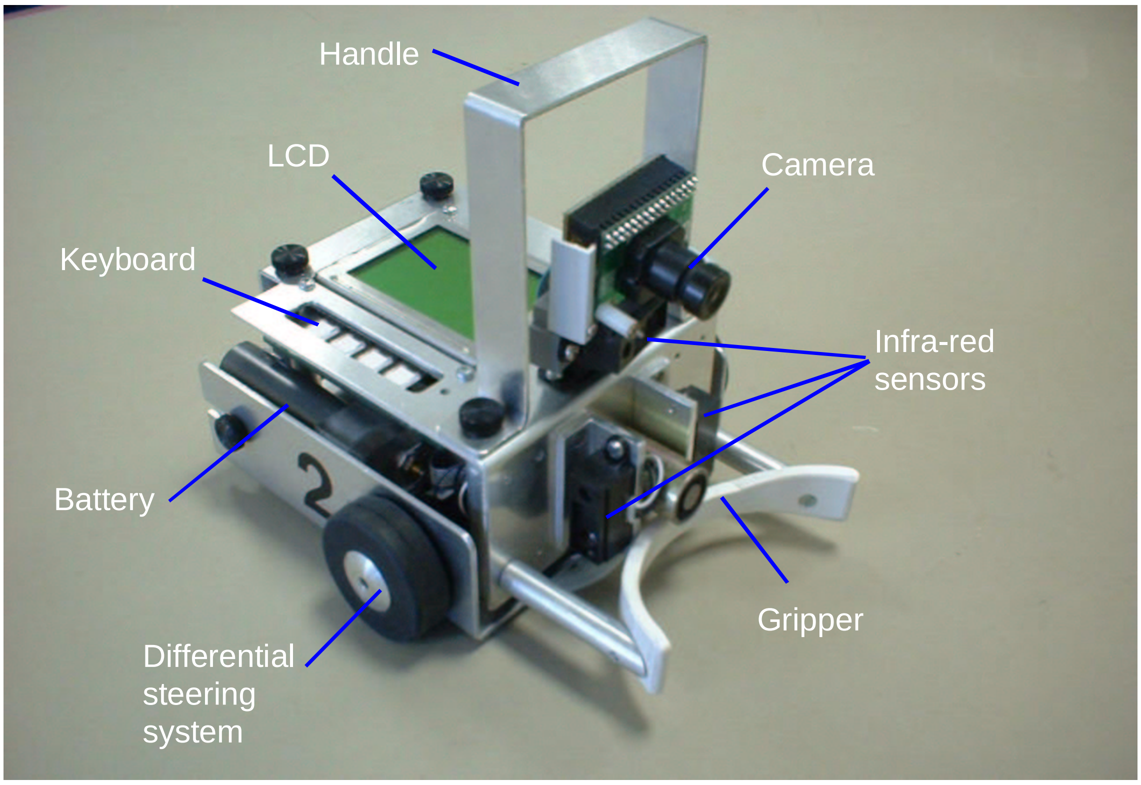


Figure 1. The EyeBot

2.2 EyeSim Simulator

EyeSim is a simulator for EyeBots. It is capable of simulating multiple robots. Most standard hardware components including camera, PSDs, radio, LCD and input buttons, and most functions of the RoBIOS library are supported. The simulator can run programs written for EyeBots without modification, they just need to be re-compiled for the simulator. However, one major constraint is that no global or static variables must be used when simulating multiple robots, as all robots would share the same variables. Beside robots the simulator only supports balls as movable objects.

The simulator can be customized using several configuration files. Possible modifications include the size of the robot, the dynamic model (e.g. collision parameters), the graphical model (how robots are graphically represented), the error model (e.g. to simulate sensor errors), and finally the world model (shape and size of the operational area).

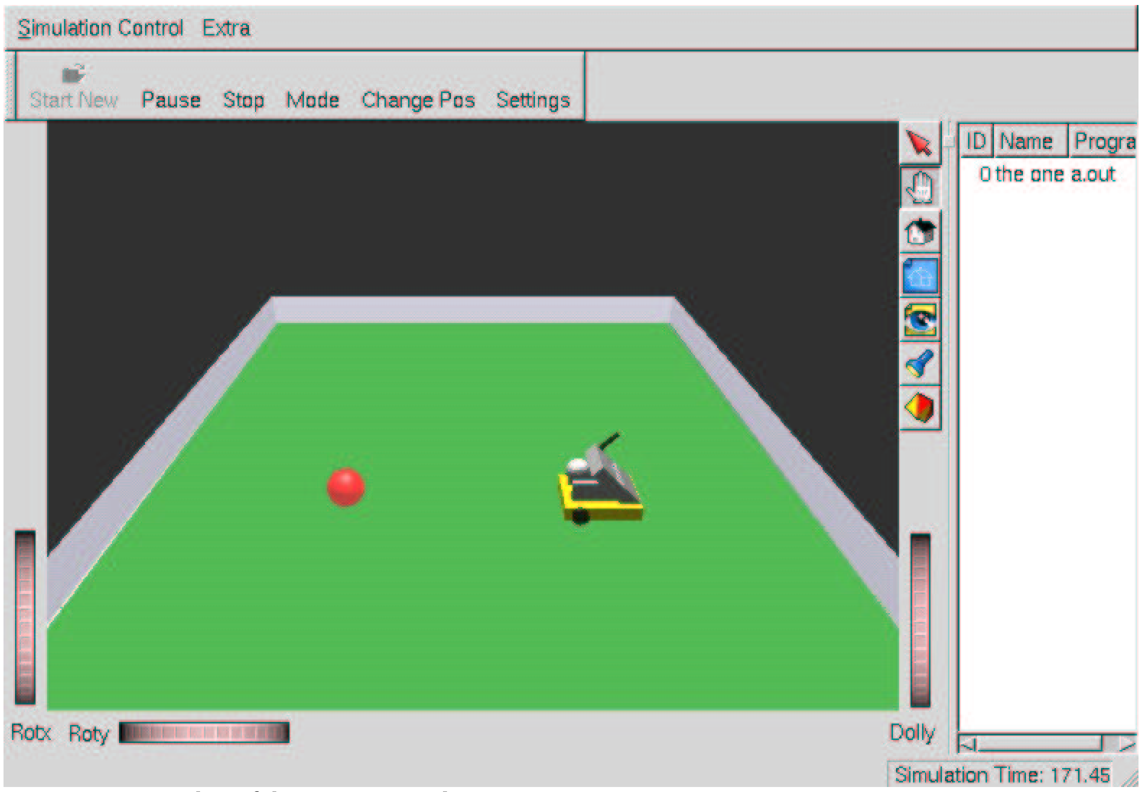


Figure 2. Screenshot of the EyeSim simulation environment

The most useful features for this project were the support for multiple balls and multiple robots, and the time laps function which sped up the testing. I adapted the parameters to reproduce the real robots that I used as accurately as possible. The changed settings include the robot size, and the positions and orientations of the PSD sensors and the camera. To simulate cubes I also increased the ball friction to prevent the balls from rolling away. However, there are some major differences to the used real robots. The currently used graphical model of the simulated robots differs considerably from the shape of the real robot. Therefore, the inclination of the camera had to be different, otherwise it would have been partly blocked by the chassis of the simulated robot. Moreover, the front PSD could not be inclined, as this arrangement is not supported by the simulator. Finally, the simulator uses a single circle for the geometrical representation of a robot, which differs significantly from the rectangular shape of the real robots.

It should be mentioned that the simulator is a new development and still in a buggy state.

2.6 Behavior-Based Robots

Early robot control systems used a model-based approach. The sensor input passed through a multitude of layers until it had any effect on the actuator output, as exemplified in Figure 3. The involved steps included complex, model-based planning and reasoning. The developed system were slow, difficult to program, little robust, and highly dependent on the correctness and accuracy of their world models, which are hard to obtain and hard to maintain.

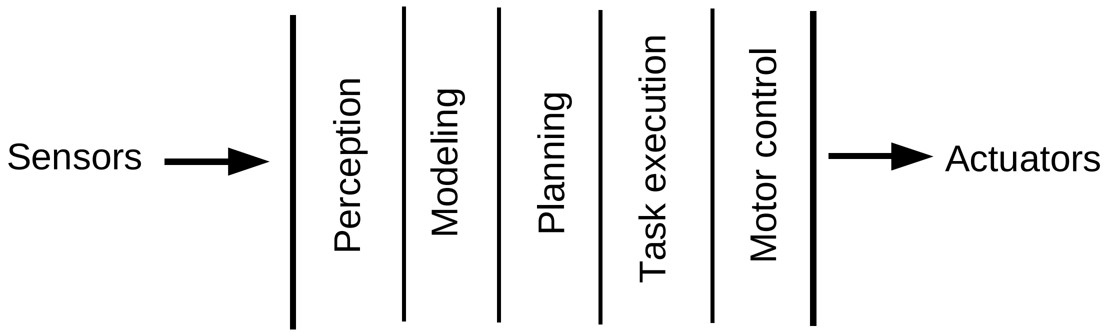


Figure 3. Schematic illustration of a model-based robot control system

In the late 1980s the researchers tried to overcome these problems by developing biologically inspired behavior-based systems. Behavior-based systems are characterized by a tight coupling of sensing and action, and the absence of world models. Behavior-based robots are also called reactive, or reflexive, as each action gets an immediate feedback.

The advantages of behavior-based systems are [9], [13]:

- Real-time performance
- Robustness in dynamic environments, as they are not dependent on world models
- Easy implementation. Each behavior is a full control program, and can thus be

designed and debugged independently. Each behavior is a specialized modules and can be kept simple.

- The system can be easily extended by adding/removing behaviors

However, the model-based approach can have an advantage over behavior-based systems in fixed environments, and generally in cases where the benefits of using a world model outweigh the disadvantages. And behavior-based system cannot be used if local sensing is not sufficient to solve the task. Remote sensor data would breach the principle of direct feedback.

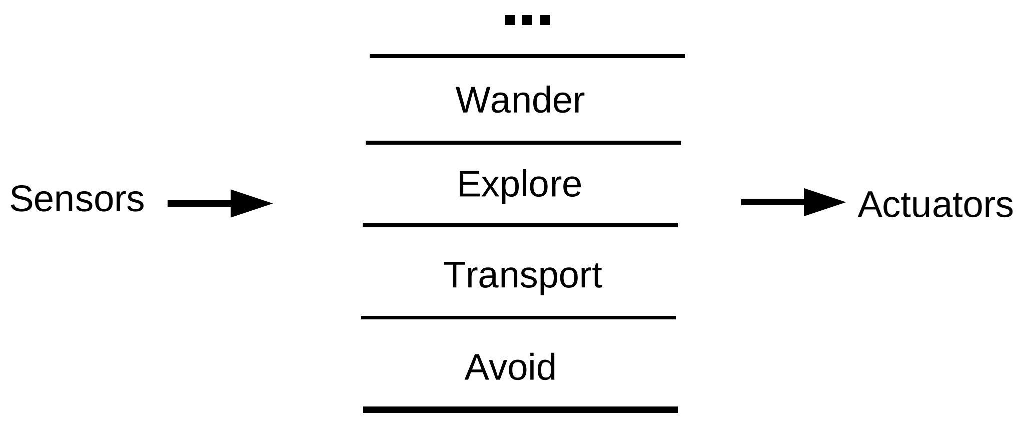


Figure 4. Schematic illustration of a behavior-based robot. In some systems all behaviors are active at the same time, and the resulting behavior is a superposition of the actuator commands. In other systems an arbiter is interposed between the behaviors and the actuators. At all times, the arbiter decides which behavior is most useful

But researchers quickly realized that entirely behavior-based robots make it hard to plan things, and perform complex tasks. The behaviors need to be intelligently coordinated and combined to achieve complex goals. Therefore, hybrid architectures were devised with low-level reflexive behaviors and high-level planning.

2.7 Multi-Agent Robotic Systems

The fast-paced progress in computer technology and robotics, accompanied by decreasing component costs, have made the development of autonomous multi-agent systems feasible. The major advantages of multi-robot systems include [12]:

- An increase in performance. The higher the parallelizability of a task, the higher the possible increase in performance by distributing it to a number of individual robots.
- The ability to solve more complex tasks. A multi-robot system may be able to perform a task that cannot be completed by an individual robot.
- Fault tolerance. A group of robots, especially of homogeneous robots, increase the redundancy of the system and thereby make it more robust to failures of individual robots. Furthermore, as the robots work as a team, it may be possible to reduce the complexity of the individual robot, and thus reduce the risk of failure.
- Distributed sensing. If the robots communicate they can share information beyond the sensor range of an individual robot.

On the other hand, the disadvantages of multi-robot systems include [12]:

- Interference. With each additional robot the communication overhead and the probability of robot collisions increase. Instead of cooperating robots may start to compete with each other.
- Failures of the communication system. By adding communication into a system it becomes less robust and the probability of system failure increases.

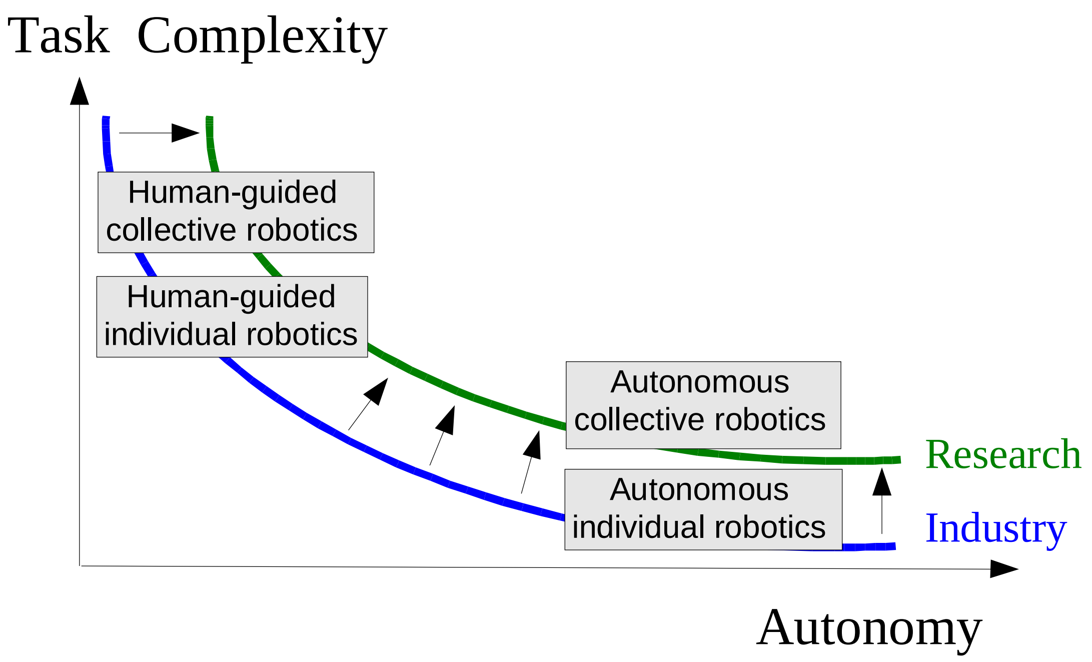

Task Complexity
Human-guided collective robotics
Human-guided individual robotics
Autonomous collective robotics
Autonomous individual robotics
Research
Industry
Autonomy

Figure 5. Multi-robot systems in research

For the problem to be solved, the development of a clustering algorithm, the benefits outweigh the disadvantages. The task is highly parallelizable, and homogeneous robots can be used. This has the potential to drastically increase the performance and robustness of the system. Attention must be paid that not 'too many' robots are active in the operational area. And the communication should be designed in way that its failure would not cause the overall system to fail.

2.8 Swarm Systems

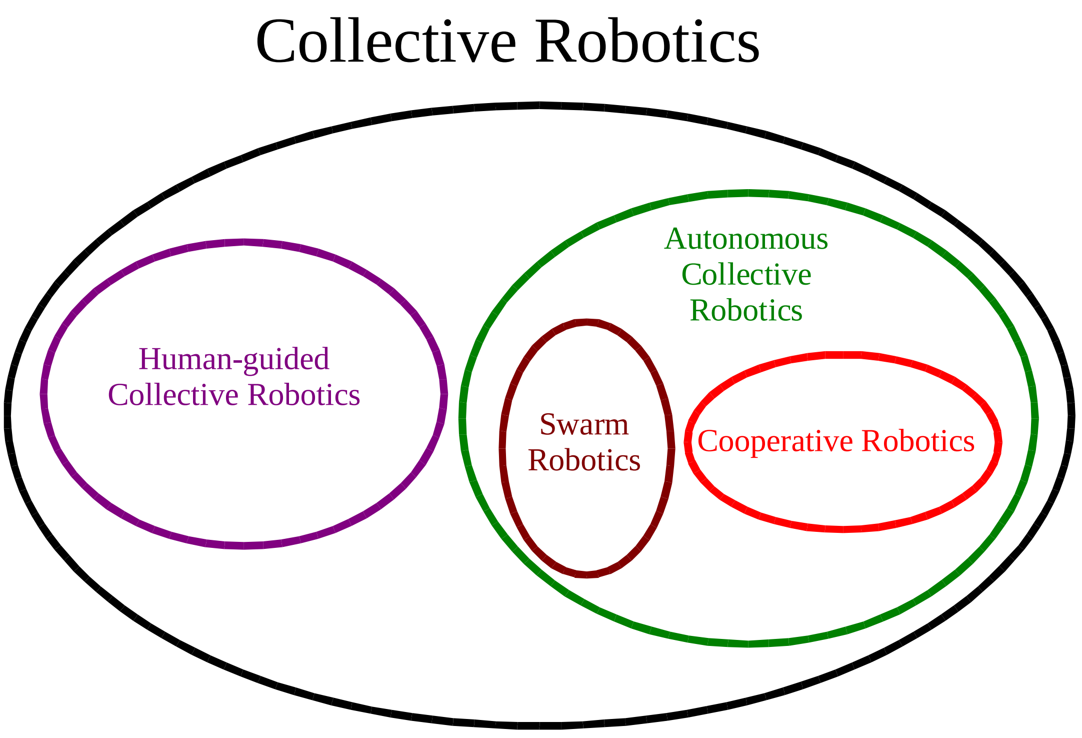

Collective Robotics
Human-guided Collective Robotics
Autonomous Collective Robotics
Swarm Robotics
Cooperative Robotics

Figure 6. Types of collective robotics

Swarm intelligence is an approach to use of behavior-based robots to create an autonomous collective robotic system [7]. It is inspired by biological systems that consist of many agents, for example ant colonies. Though colonies consist of stereotypical, unreliable and simple agents, as a whole they are capable of accomplishing complex tasks in dynamic and varied environments. The behavioral repertoire of the agents is limited, they follow simple rules and use simple local communication. A global structure emerges from the actions of many, without central control or coordination. Swarm Intelligence relies on distributedness and decentralization, simple and specialized agents, direct or indirect basic interactions among those agents and robustness to failure of individuals. In this project the principles of Swarm Intelligence were to be applied to the problem of cube clustering using multiple autonomous robots.

2.5 Collaborative Clustering

The basic principle of collective clustering algorithms is very simple. Beckers, Holland and Deneubourg showed that

1. if robots have the means of moving some discrete items,
2. are able to make clusters,
3. and have some way to estimate local density,

then, with the influence of noise and stochastic robot-robot and robot-environment interactions, in the end a single cluster will remain [7].

Basically the reasoning is as follows: If the probability of leaving a cube on a cluster increases with the cluster size, and the probability of taking a cube from a cluster decreases with the cluster size, then the rate of growth will increase with the size of the cluster. As the rate of growth over all clusters is zero (the number of cubes is constant), the result must be positive growth for the largest clusters and negative growth for the smallest ones. Thus, N clusters will become (N-1) clusters, and so on.

2.9 Other Clustering Experiments

First clustering experiments were conducted by Beckers, Holland, and Deneubourg in 1994 [14]. The used robots were entirely behavior-based, and IR sensors were the only source of input. They demonstrated that clustering could be done with minimalist robots. In 1998 Holland and Melhuish used more sophisticated robots with IR sensors, grippers, and optical sensors for color detection [7]. Holland and Melhuish tried to mimic the behavior of Leptothorax ants and their concentric annular sorting. They also performed more complex experiments including the usage of probabilistic algorithms. In 2000, Burkhard Iske and Ulrich Rückert incorporated local communication in their clustering algorithm [8]. The infra-red sensors of their Khepera robots were used to communicate the age of cluster points. The oldest one of them emerged as global cluster point.

2.3 HSV Color System

HSV stands for hue, saturation and value.

The hue describes where a color lies along the spectrum. Hue values are organized in a color circle, with red at 0 degrees, yellow at 60 degrees, then green, cyan, blue, and finally magenta at 300 degrees.

The saturation describes how pure a color is. The saturation value goes from 0% (gray) to 100% (maximum purity). A low value results in a neutral, dull color, whereas a high value means a strong, pure color.

Value, or brightness. A value of 0% means completely black, while 100% is the brightest value that a color can have. A maximum value does not mean white, unless the saturation is zero. A maximum value is simply the brightest value a color can have at a particular saturation.

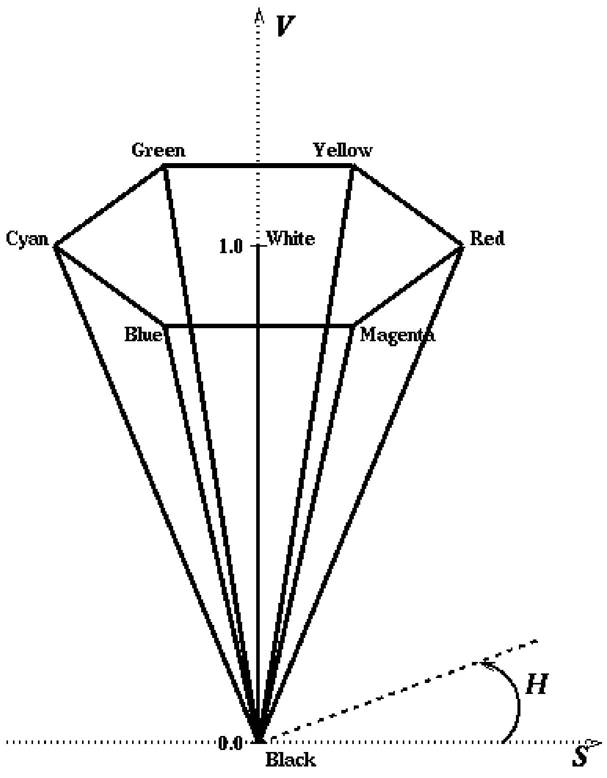


Figure 7. The HSV cone

The volume spanned up by the three parameters of the HSV color system can be viewed as an inverted cone. Each point in that cone represents a color in the HSV system. The hue is the angle in a plane orthogonally to the cone's symmetry axis. The saturation is the radius, the distance of the point from the symmetry axis. Finally, the value is the coordinate along the symmetry axis of the cone, it describes the height of the point.

2.4 RGB Color System

RGB stands for the three basic colors used in the RGB system: red, green and blue. The complete color scale originates through the superposition of these three colors. Thus, in the RGB model, every color is represented as a set of three independent values: a value for red, a value for green and a value for blue. In the EyeCam images each value can range from 0 to 255. If all three values are 0, the resulting color is black; if all three values are 255, the color is white. Around 16 million colors (256^3 = 16,777,216) can be represented.

In the RGB system a color can be imagined as point in a cube spanned up by the three basic colors.

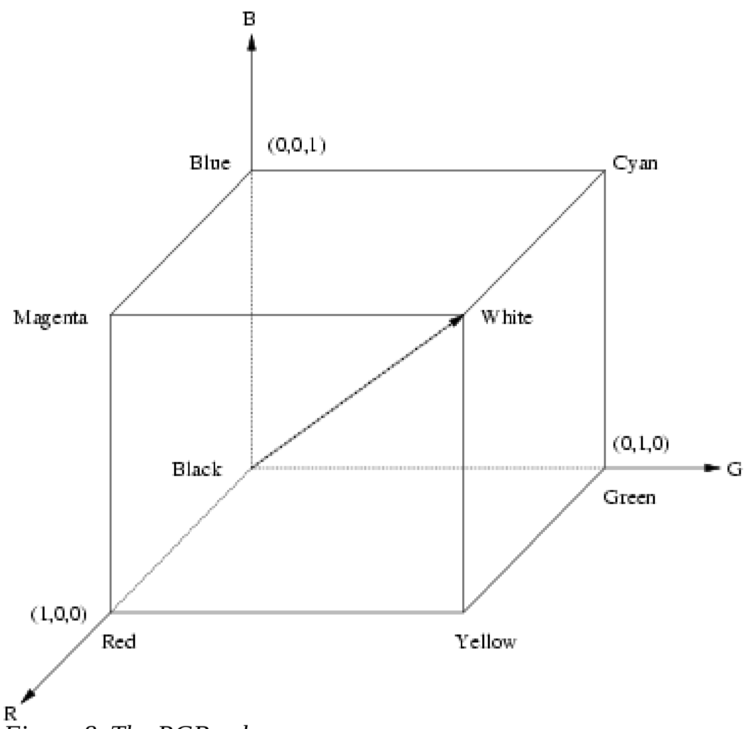


Figure 8. The RGB cube

3. System Requirements

The objective was to develop the software for a behavior-based robotic system. The system should make use of the main characteristics of swarm intelligence to collectively cluster randomly placed cubes in a walled area. The software was to be implemented for a set of autonomous mobile robots. The following requirements were derived (Note: these are extracts from the system requirements specification, see the original document for details).

Requirements regarding the swarm:

- The robot colony is to be scalable. It should be possible to add or remove robots at any time without disruption of the overall swarm behavior
- That includes in particular that the robot colony should be robust regarding the failure of individual robots
- The control of the robot colony should be decentralized. No master individuals and no central coordination should be required
- The robot colony is to be redundant in the sense that the robots should be homogeneous and each robot should have full decisional power
- There should be some form of implicit or explicit communication among the robots to ensure a common cluster point

The behavior-based agents should have the following characteristics:

- Fully autonomous and behavior-based individuals
- The agents should be reactive, have a tight coupling of sensing and action
- Use local sensing and perform only local action
- Use image processing for object detection
- It is not required for the robots to be adaptive

Nonfunctional Requirements:

- The developed software should work properly both within the simulation environment and in a real environment
- All cubes should be collected, and a single cluster point should remain at the end

Functional Requirements:

- The robots should collectively cluster randomly distributed cubes on a plane
- The robots should be able to recognize and distinguish cubes, robots and walls
- The robots should explore the plane to find cubes
- The robots should push found cubes to a cluster point
- The robots should be able to avoid obstacles (walls and possibly other robots)
- There should be some form of implicit or explicit robot-robot communication to determine a common cluster point

4. System Model

The system model is the result of a system analysis. Based on the requirements specification it aims to describe the desired system behavior in a consistent, complete, realizable and verifiable way [11].

State chart diagrams were used to represent the system model. The system model contains three different versions. This is to reflect three different approaches to the robot communication.

Case 0:	Implicit communication A common cluster point emerges through stochastic robot-robot and robot-environment interactions (see 2.5)
Case 1:	Explicit communication by broadcasting A common cluster point is determined and communicated by broadcasting to nearby robots
Case 2:	Explicit one-on-one communication A common cluster point is determined and communicated by one-on-one data exchanges between robots that cross each other's paths

For clarity and comprehensibility the system model was separated into eight diagrams. A short description of the diagrams follows:

1. *Overview*
   This diagram gives an overview of the system behavior. The seven states in this diagram are super-states; they are specified in the following seven remaining sub-diagrams.

2. *User Interface*
   This diagram describes the reactions of the system to user input.

3. *Exploring,*

4. *Pushing,*
5. *Avoiding,*
6. *and Communicating* (only in case 2)

   These four diagrams describe the exploring, pushing, avoiding or communicating behavior of the robot, respectively. There are transitions between these four diagrams. Pseudo-states named 'Entry to ...' or 'Exit to ...' (e.g. 'Entry to Pushing' or 'Exit to Avoiding') are used to indicate a transition from one diagram to another. This was necessary to make the transitions easier to trace.

7. *Broadcasting* (only in case 1)

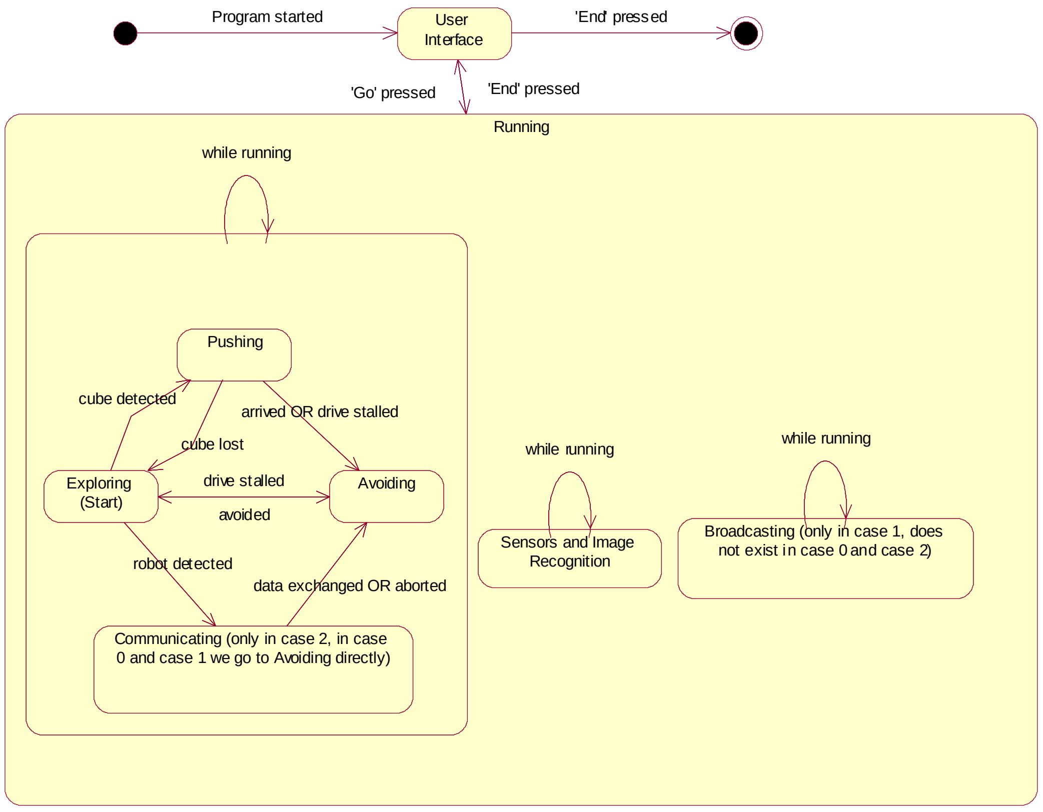
   This diagram describes the broadcasting process of the robots.

8. *Sensors and Image Recognition*

   Finally, this diagram describes the sensing and image recognition processes in the robots.

4.1 Overview Diagram

This diagram gives an overview of the system behavior. The seven states in this diagram are super-states, they are specified in the following seven remaining sub-diagrams.



At every state that involves movement we have to check for cubes, robots, walls, proximity to cluster point and stalled motors.

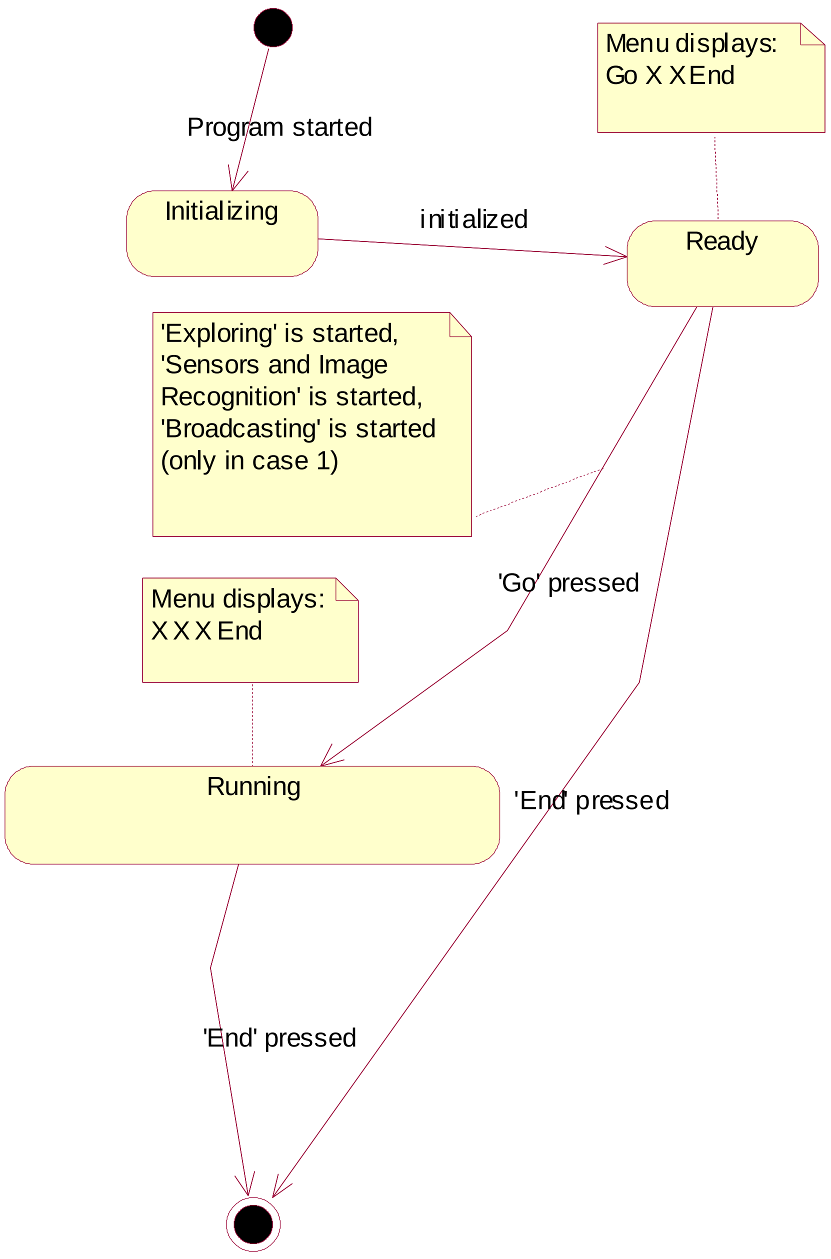
First the clustering is done only through estimation of the cluster sizes (So we use implicit communication, the robots tend to push towards larger clusters). Then the clustering is done with communication through broadcasting. Finally with one-on-one communication.

case 0: Implicit communication
case 1: Broadcasting (requires same starting point)
case 2: One-on-one communication

In case 1 and 2 the position with the highest known cluster point density is communicated.

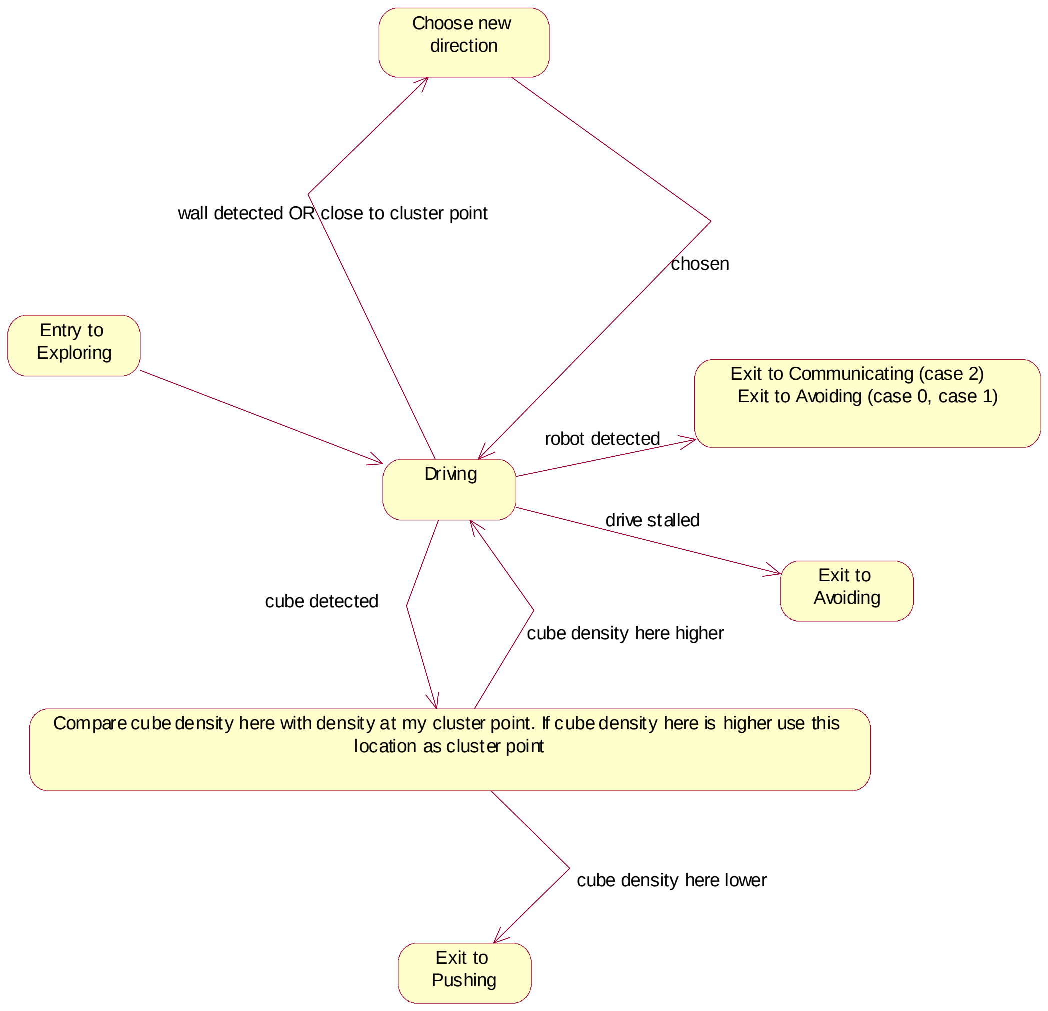
4.2 User Interface Diagram

This diagram describes the reactions of the system to user input.



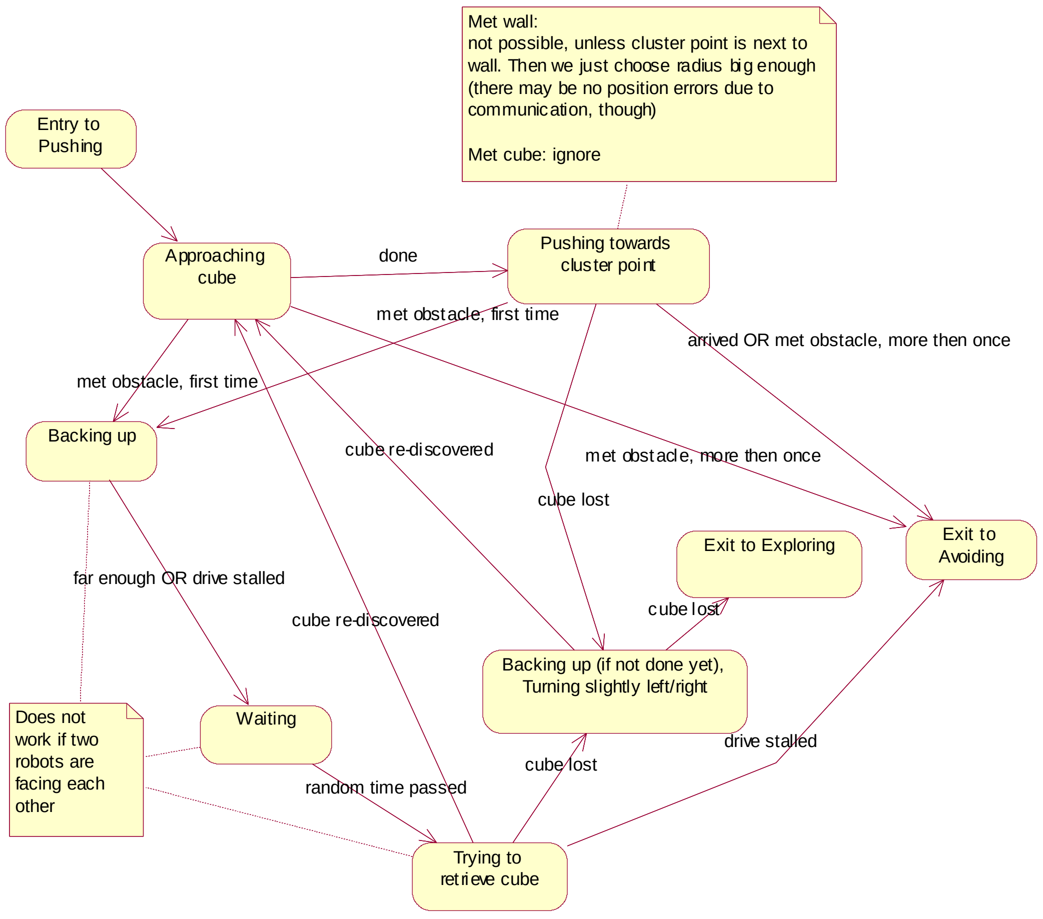
4.3 Exploring Diagram

This diagram describes the exploring behavior of the robot.


Choose new direction
wall detected OR close to cluster point
chosen
Entry to Exploring
Exit to Communicating (case 2)
Exit to Avoiding (case 0, case 1)
robot detected
Driving
drive stalled
Exit to Avoiding
cube detected
cube density here higher
Compare cube density here with density at my cluster point. If cube density here is higher use this location as cluster point
cube density here lower
Exit to Pushing

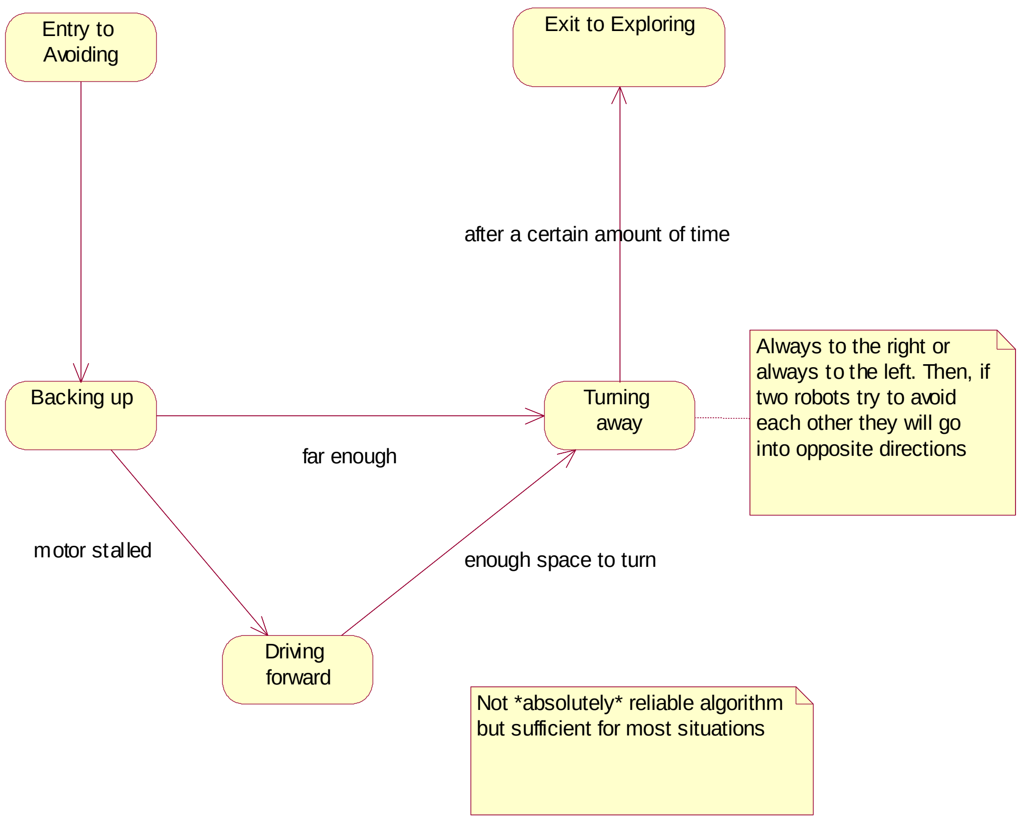
4.4 Pushing Diagram

This diagram describes the pushing behavior of the robot.


Met wall:
not possible, unless cluster point is next to wall. Then we just choose radius big enough (there may be no position errors due to communication, though)
Met cube: ignore
Entry to Pushing
Approaching cube
done
Pushing towards cluster point
met obstacle, first time
arrived OR met obstacle, more then once
met obstacle, first time
Backing up
cube re-discovered
met obstacle, more then once
cube lost
Exit to Exploring
Exit to Avoiding
far enough OR drive stalled
cube re-discovered
cube lost
Backing up (if not done yet), Turning slightly left/right
Does not work if two robots are facing each other
Waiting
drive stalled
cube lost
random time passed
Trying to retrieve cube

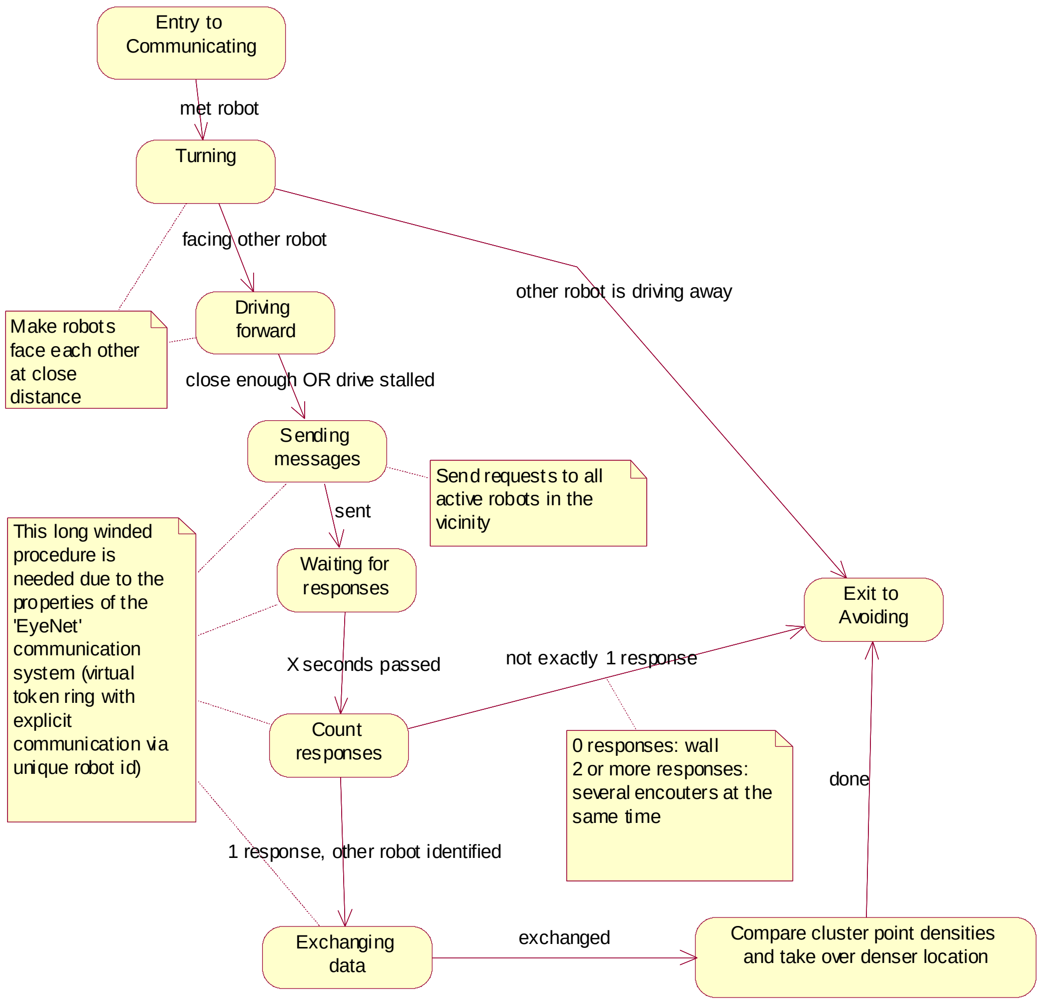
4.5 Avoiding Diagram

This diagram describes the avoiding behavior of the robot.


Entry to Avoiding
Exit to Exploring
after a certain amount of time
Backing up
Turning away
far enough
Always to the right or always to the left. Then, if two robots try to avoid each other they will go into opposite directions
motor stalled
enough space to turn
Driving forward
Not *absolutely* reliable algorithm but sufficient for most situations

4.6 Communicating Diagram

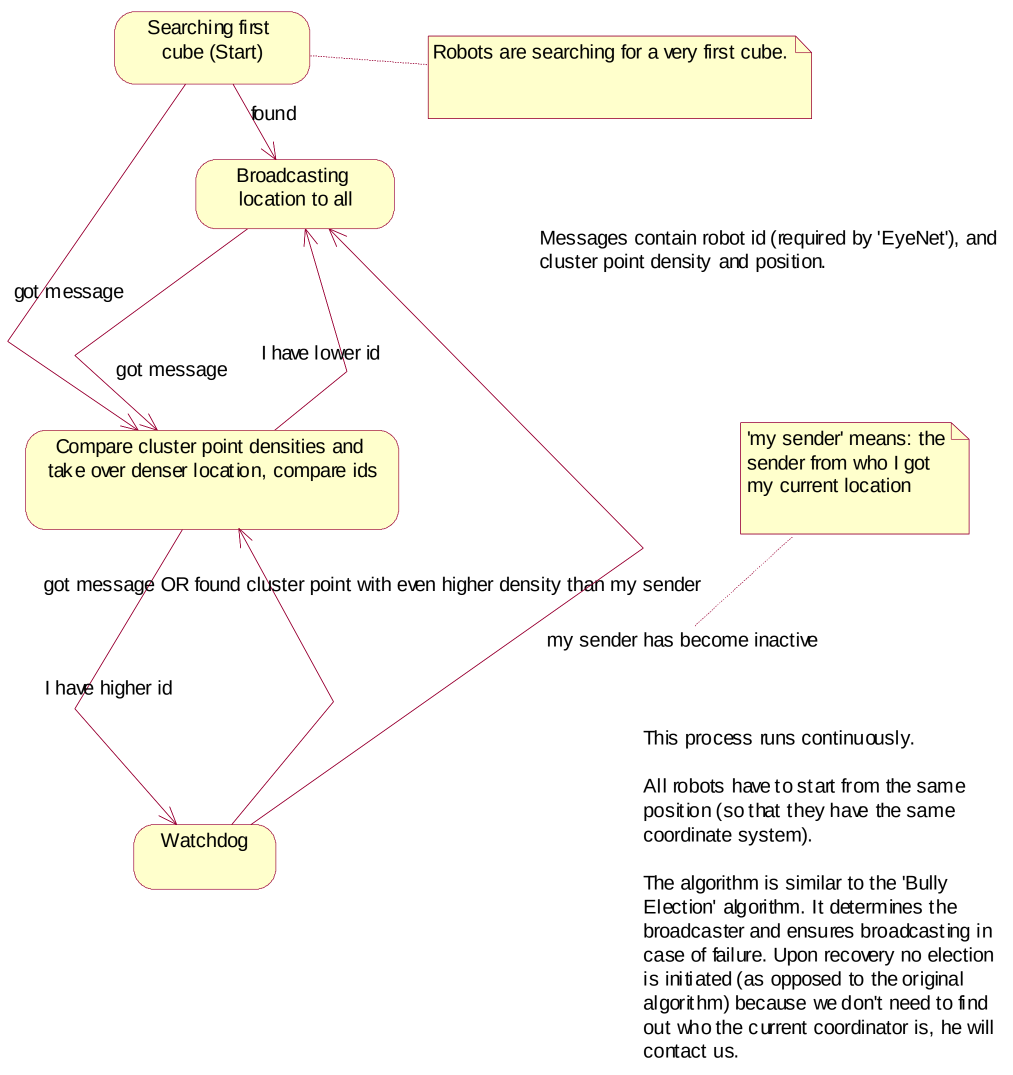
This diagram describes the communicating behavior of the robot.


Entry to Communicating
met robot
Turning
facing other robot
other robot is driving away
Make robots face each other at close distance
Driving forward
close enough OR drive stalled
Sending messages
Send requests to all active robots in the vicinity
sent
This long winded procedure is needed due to the properties of the 'EyeNet' communication system (virtual token ring with explicit communication via unique robot id)
Waiting for responses
Exit to Avoiding
X seconds passed
not exactly 1 response
Count responses
0 responses: wall
2 or more responses: several encouters at the same time
done
1 response, other robot identified
Exchanging data
exchanged
Compare cluster point densities and take over denser location

Messages contain robot id (required by 'EyeNet'), and cluster point density and position.

4.7 Broadcasting Diagram

This diagram describes the broadcasting process.


Searching first cube (Start)
Robots are searching for a very first cube.
found
Broadcasting location to all
Messages contain robot id (required by 'EyeNet'), and cluster point density and position.
got message
got message
I have lower id
Compare cluster point densities and take over denser location, compare ids
'my sender' means: the sender from who I got my current location
got message OR found cluster point with even higher density than my sender
my sender has become inactive
I have higher id
Watchdog
This process runs continuously.
All robots have to start from the same position (so that they have the same coordinate system).
The algorithm is similar to the 'Bully Election' algorithm. It determines the broadcaster and ensures broadcasting in case of failure. Upon recovery no election is initiated (as opposed to the original algorithm) because we don't need to find out who the current coordinator is, he will contact us.

Note:

Communication by broadcasting is just an evolutionary step towards case 2 (local communication). As all robots have to start from the same point, they could just push all cubes to their common starting position. There would be no need for any communication. However, the implementation of this case helps to become familiar with the features of the radio communication system, and experience for the implementation of case 2 is gained.

The broadcasting process and the election algorithm that it uses might be regarded as breach of the requirements N210 ('no central coordination') , N310W ('no initialization') and N112 ('local action'). I want to address these concerns here.

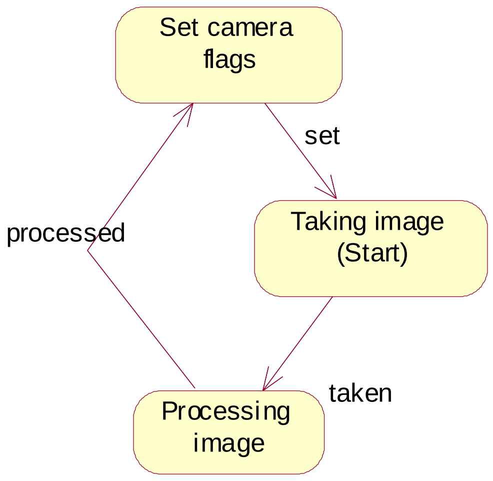
N120: The election algorithm is only needed to ensure that no superfluous communication traffic occurs. The communication process could be easily modified such that no election would be necessary and every robot would keep broadcasting to other robots in its vicinity.

N112: As the range of broadcasting is limited to the close vicinity of a robot, this can still be regarded as local action.

N310W: For broadcasting all robots need to have the same starting point. From the user effort point of view this poses no real argument for not starting all robots from the same position. It is correct, though, that for various reasons it might be desirable to start a robot from any point on the plane.

4.8 Sensors and Image Recognition Diagram

Finally, this diagram describes the sensing and image recognition processes in the robots.



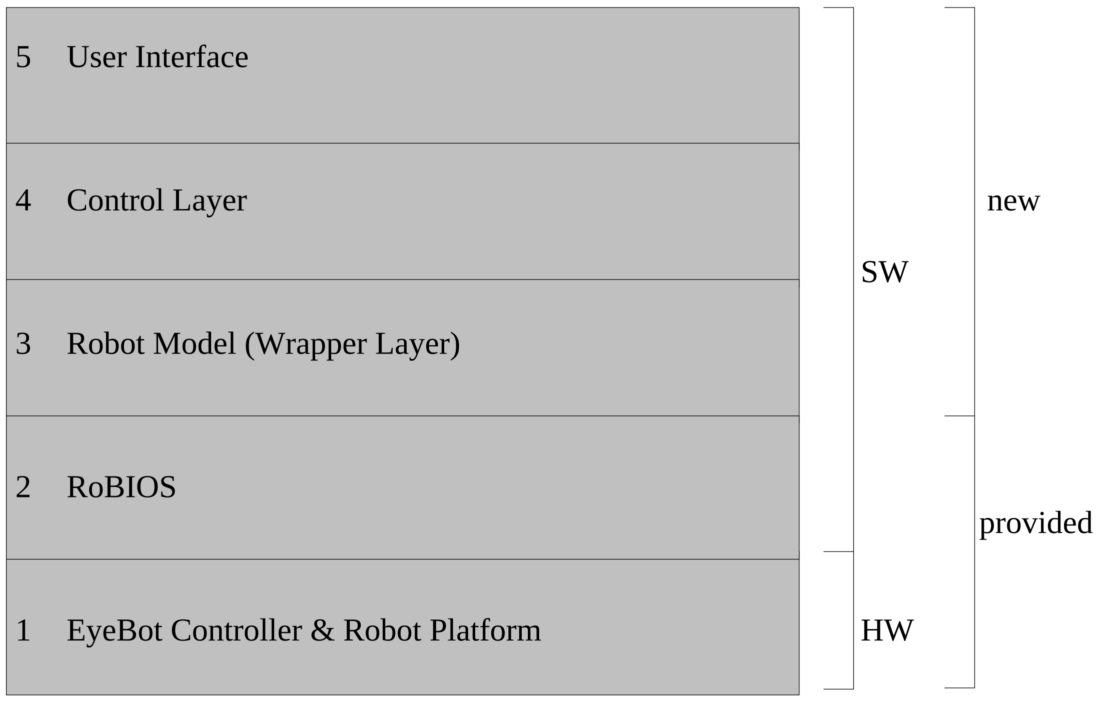
This process runs continuously.

5. System Architecture

The system architecture is based on the system model. It constitutes the transition from system analysis to design. The system architecture contains the decomposition of the overall system into sub-systems and components, and describes the interfaces among them [11].

5.1 Fundamental Design Decisions

I decided to divide the system into five layers:


5 User Interface
4 Control Layer
3 Robot Model (Wrapper Layer)
2 RoBIOS
1 EyeBot Controller & Robot Platform
SW
HW
new
provided

Layer 1: Hardware layer

This layer represents the EyeBot controller and the robot hardware platform.

Layer 2: RoBIOS layer

This layer contains the RoBIOS library functions.

Layer 3: Wrapper layer

Layer 3 contains wrapper classes. These classes fulfill four functions: Firstly, they encapsulate the needed C functions from the RoBIOS library in C++ classes. Secondly, they constitute a layer of separation between the program and the provided libraries, making it easier to adapt the program to changes in the underlying libraries. Thirdly, the classes can combine low-level library functions to provide high-level functionality to and hide complexity from the layers above. Finally, the classes provide the opportunity to use consistent naming conventions throughout the newly developed software.

Layer 4: Control layer

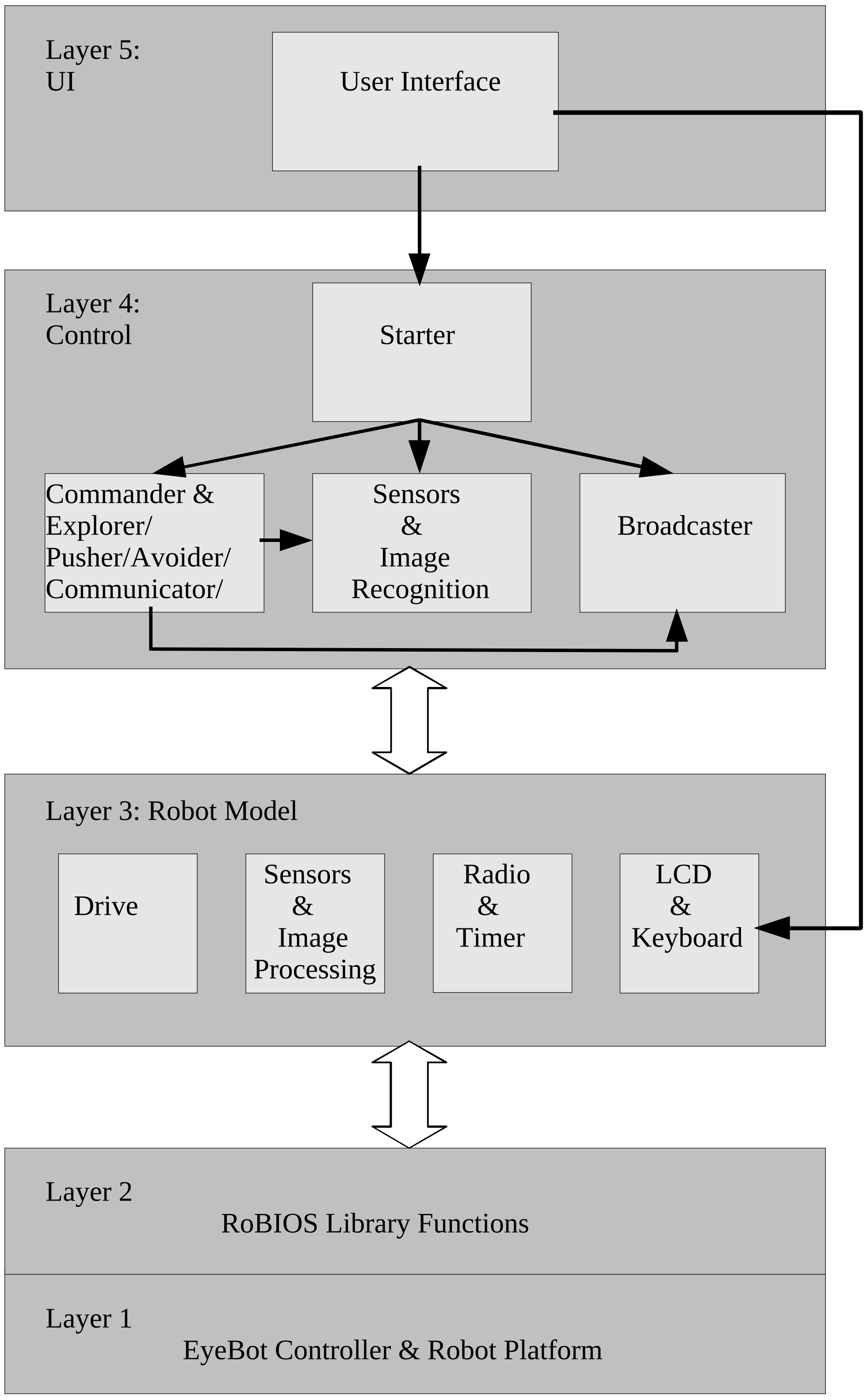
Layer 4 contains the main functionality. That includes all classes that are needed to control the clustering from a high-level perspective. The classes in layer 4 use the wrapper classes in layer 3 to control and communicate with the robot. A point of entry for the user interface of layer 5 is provided.

Layer 5: User interface layer

Layer 5 contains the user interface. It separates the user interface from the functional part of the program.

5.2 Diagram of the Software System Architecture

The following diagram gives a basic overview of the system architecture.



Legend:

→	(specified) association
⇕	indicates unspecified associations between layers

Note: The Broadcaster only exists when using global communication. The Communicator only exists when using local communication. See system model for details.

The program consists of up to four threads. One thread for the User Interface, one thread for the Sensors and Image Recognition (SIR), one thread for the behaviors (Commander & Explorer/Pusher/Avoider/Communicator) and finally, if using global communication, one additional thread for the broadcasting process. This partitioning ensures responsiveness to user input and reactivity of the behaviors. The number of threads is kept to a minimum to avoid overhead due to context-switches. But it is still ensured that the time-consuming image recognition and broadcasting processes do not block the UI and behaviors, that have short lead times but need to be called regularly.

Currently four behaviors are realized, exploration, pushing, avoidance and communication. The communication behavior is only activated when using local communication. As we use a behavior-based approach, the control over the robot can be handed over to a single class that represents one behavior.

The Commander switches between the behaviors choosing the most appropriate one based on the result code returned by the last activated behavior. Activated behaviors are not pre-empted, they return when they have served their purpose or when they recognize that they cannot cope with the situation. With this framework, behaviors can be easily added or removed, and the sequence of behaviors can be easily changed.

The Starter is just used to provide an easy access point for the user interface by hiding complexity. It enables and disables the threads that form the clustering algorithm making the user interface independent from functional details.

The SIR provides high-level environment information to the behaviors. It runs in a own thread to prevent it from blocking other processes.

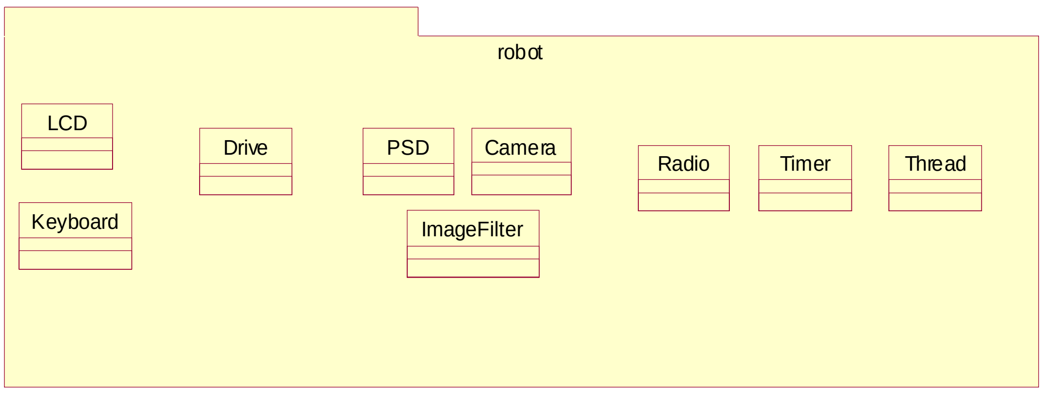
The Broadcaster realizes the global communication process when using communication by broadcasting.

5.3 System Components

The program consists of two packages. A robot package containing all classes of the wrapper layer and a clustering package that contains all classes of both the control layer and the user interface layer.

This section gives a brief overview of the system components. A fully commented and browsable documentation is available (see components specification).

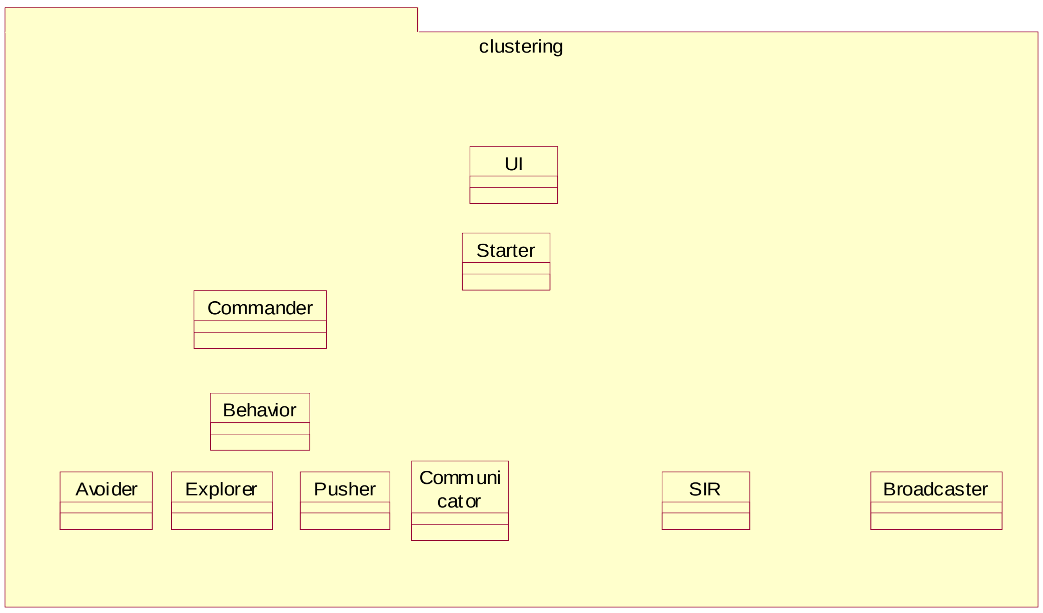
The robot package has the following classes:



LCD:	This class provides methods to access the robot LCD.
Keyboard:	This class provides methods to read from the robot keyboard.
Drive:	This class provides methods to control the robot drive.
Camera:	This class provides methods to access the robot camera.
ImageFilter:	This class provides image filters.
PSDs:	This class provides methods to access the infra-red sensors.
Radio:	This class provides methods to use the radio communication.
Timer:	This class provides methods to access the robot timer.
Thread:	Base class from which all multi threaded classes are derived. The derived classes inherit a set of methods that can be used to control the thread. Once the thread is initialized and made ready, its `run` method is scheduled for

execution. The `run` method is declared as virtual and has to be overwritten by the inheritor.

The clustering package contains the following classes:



CubeClustering (not in diagram, it is not a class):

This component contains the `main` function of the program. It initializes and starts the program.

UI: This class implements the user interface. It allows the user to start and stop the clustering using the input buttons of the EyeBot controller.

Starter: This class provides a point of entry for the user interface to control the clustering algorithm. This ensures that the functionality is separated and hidden from the user interface. The class provides a few public methods to start and stop the clustering. Starter controls the state of the threads (ready, suspended, ...) that form the clustering algorithm

Commander: Activates and switches between the behaviors of the robot (current implementation: exploring, pushing, avoiding and communicating. Communicating only when using local communication).
The decision which behavior to activate is based on the current state of the commander and the return code it receives from the last activated behavior. Once a behavior has served its purpose it will return with a result code. The commander does not actively interrupt activated behaviors.

Behavior: Base class from which all behaviors are derived. This provides them with the same base properties and makes adaptations easier, that are to affect all behaviors.

Explorer: Explores the plane using simple search patterns. Avoids walls and the own cluster point when exploring. Returns when a cube or a robot has been detected.

Pusher: This class implements the pushing behavior of the robot. The robot approaches the cube and tries to push it to its cluster point. If the robot meets other robots on its way it tries to avoid them.

Avoider: Implements the avoiding behavior of the robot. The robot tries to back up and then turns away from the obstacle.

Communicator: This class implements the local communication behavior of the robot. It is only needed when using local communication. First the robot is justified so that it faces the other robot at a close distance. Then it tries to identify the other robot. If successful their cluster points are exchanged and the denser of the two cluster points is chosen as common cluster point.

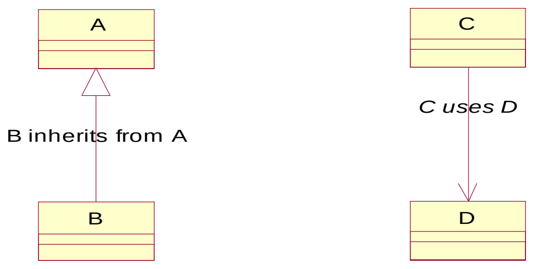
SIR: This class (Sensor and Image Recognition) provides high-level information to the behaviors by pre-processing the data internally. The provided information includes current obstacle distances as received by the infra-red sensors. But in particular information about the locations of cubes and robots that are visible on camera are provided

Broadcaster: This class implements the broadcasting abilities of the robot. It is only needed when using communication by broadcasting. The broadcasting algorithm makes sure that a cluster point is determined and communicated to all robots in the vicinity. It also makes sure that the cluster point position is retained should the broadcasting robot fail. See system model for details.

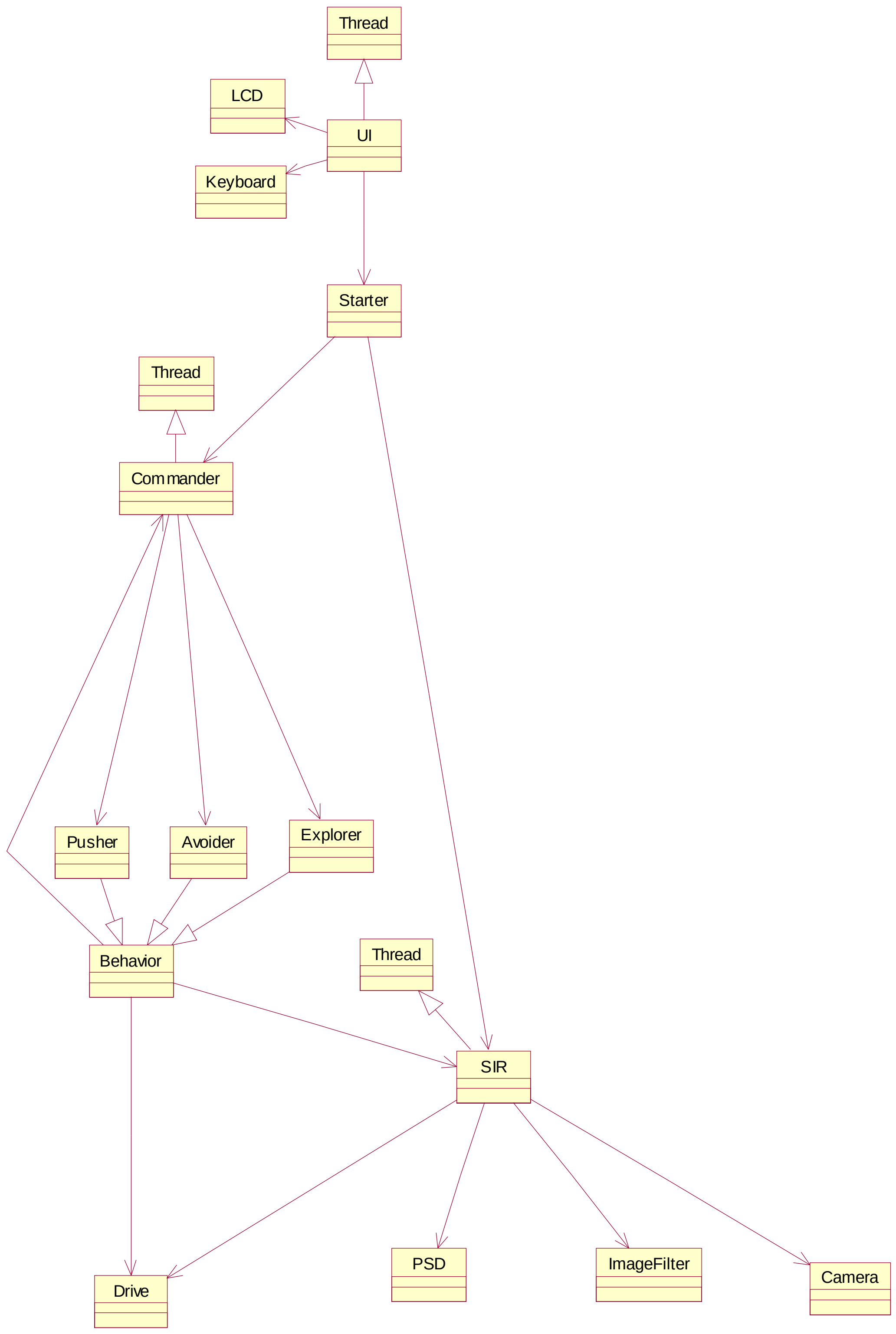
5.4 Class Diagrams

The following class diagrams show the relationships between the classes of the layers 3 to 5.

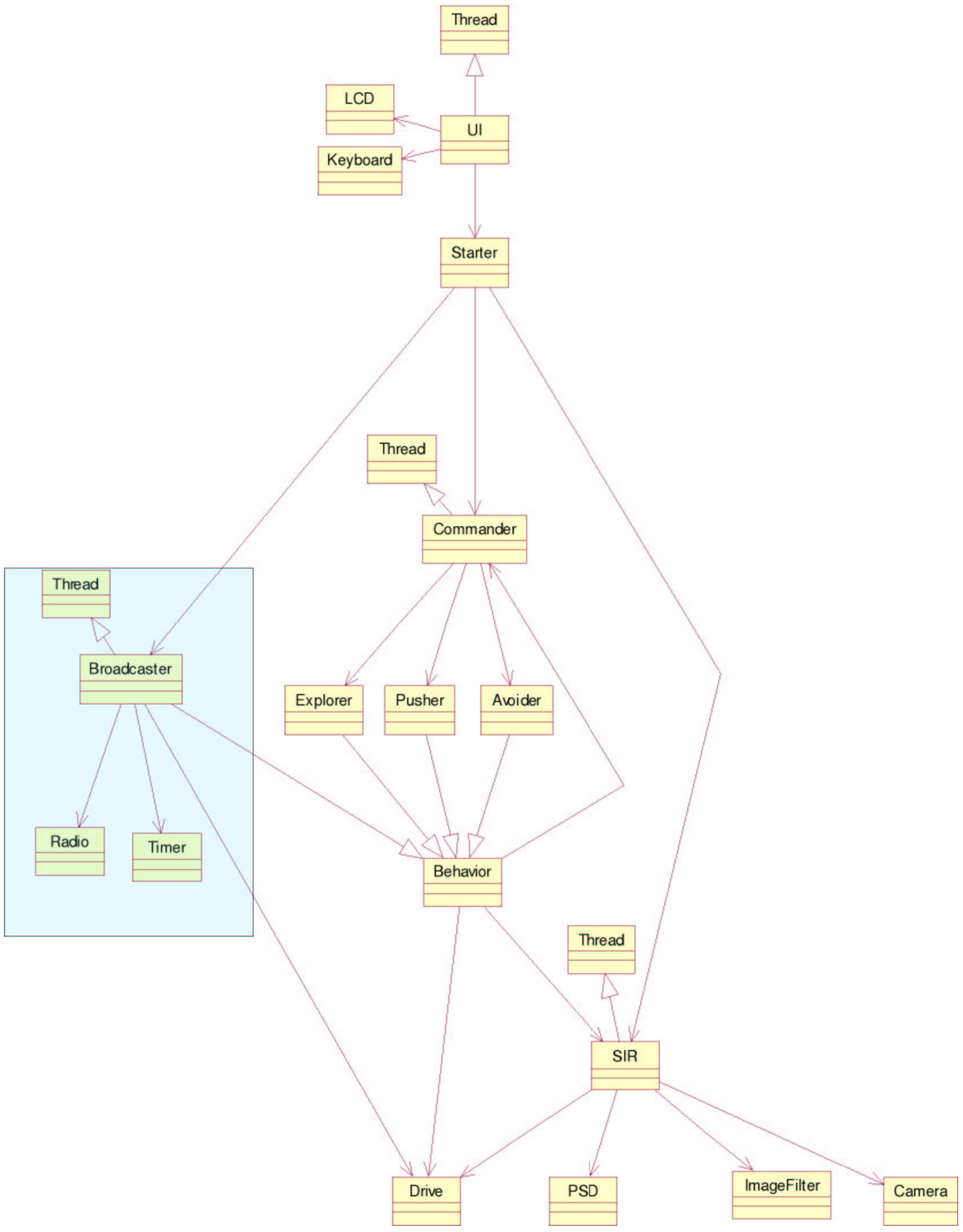
Legend:



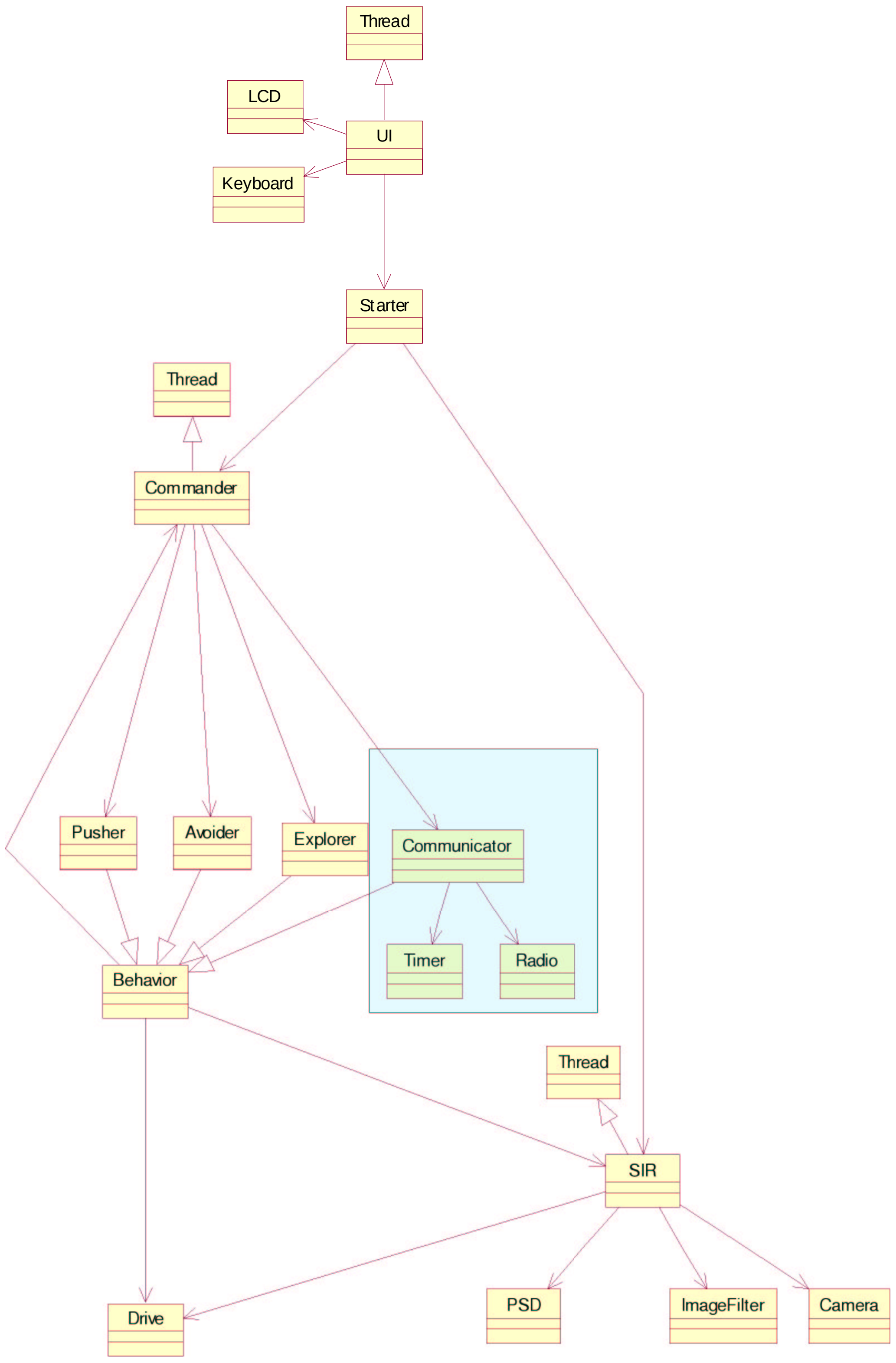
5.4.1 Implicit Communication



5.4.2 Communication by Broadcasting



5.4.3 Local Communication



6. Implementation

6.1 General Implementation Decisions

There are two main sources of input, the camera and the infra-red sensors. I decided to use the camera for cube detection, and the PSD sensors for obstacle avoidance. The red cubes are well distinguishable from the environment on the color camera. The PSD sensors, on the other hand, quickly deliver accurate distance data of surrounding obstacles. A final source of input, the information whether the drive is stalled, is used for obstacle avoidance as well.

Of all behaviors, avoidance has the highest priority followed by pushing, communication and finally exploration. This ordering prevents the robot from becoming stuck and ensures that even with high robot densities the actual clustering does not come to a stop due to excessive robot-to-robot communication.

I decided to develop the software for real robots first, and then to port the program to the simulator. The demands regarding robustness are usually higher for real environment systems because of errors in sensors and actuators. A system running perfectly on the simulator may fail in real-life. The other way round, a failure is less likely.

I decided to express angles in degree instead of radian, mainly because it is easier for the programmer to visualize and handle angles in degree. As the RoBIOS functions use radian, the Drive class needs to perform a few conversions. But the conversions are restricted to the Drive class only. The high-level classes, mainly the behaviors, can use degree throughout.

6.1.1 Coordinate systems

The robots use two coordinate systems for navigation. A global coordinate system, and a local one. The local coordinate system is 'carried along' by the robot. The origin moves around with the robot's center. The x-axis always points to the current direction of the robot. The global coordinate system is set when the robot drive is initialized. The origin is the starting point of the robot, and the positive x-axis shows the orientation of the robot at startup.

Whenever possible I used the local coordinate system. The motion commands are easier to understand and program, and the computations are faster when calculating entirely with local coordinates.

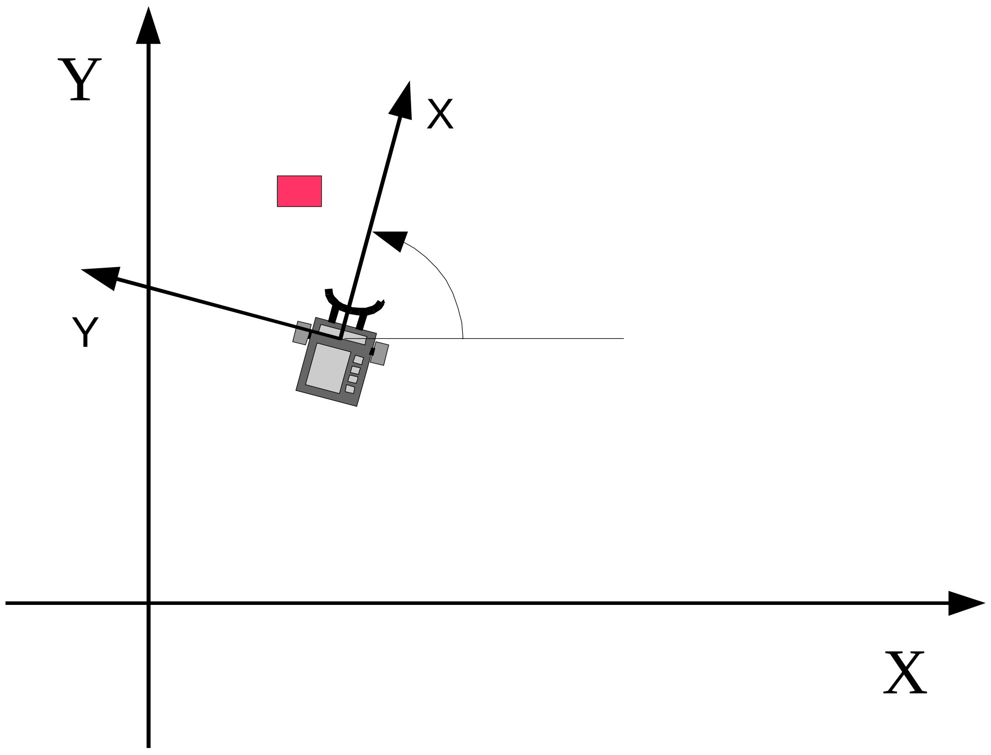


Figure 9. Local and global coordinate system

The cluster point of a robot is always the origin of its global coordinate system. It is easier for the programmer to make to robot push a cube to the origin of its coordinate system, and the computations are faster. And, as we will see later, the usage of the origin as cluster point makes the synchronization of coordinate systems easier. So, when a robot adopts a new cluster point, it introduces a new global coordinate system. The origin of the new coordinate system is at the new cluster point location. Then the robot calculates its own position in the new system.

6.1.2 RGB to HSV

One major advantage of the HSV color coding over the RGB system is the invariance of the hue value with regard to illumination and camera orientation [5]. The hue of the red cubes to be collected are distinct from the hues of all other objects in the operational area, according to the requirements specification. Therefore we can use the hue value to reliably identify cubes. This approach also ensures a high performance as only one value instead of three needs to be considered. For these reasons I decided to use the HSV color system for image recognition. The RGB images obtained from the EyeCam are converted into HSV values. For each pixel in the image we calculate the hue from the three RGB values. In the current implementation 253 hues can be distinguished. The hue value ranges from 0 to 252, the value 255 is used for objects with no hue. The saturation and brightness are not calculated as they are not needed.

6.2 Behaviors and Image Recognition

6.2.1 Detection of Cubes

I decided to use a hue-based detection method, based on the experiences of Birgit Graf [1]. The cubes have a distinctive color, making a color-based detection method the obvious choice. The HSV color system has the advantage that the hue value is invariant to illumination and camera position [5]. The 24-bit RGB image provided by the EyeCam is converted to an HSV representation on-the-fly. Only the hue values are calculated, as they are sufficient to reliably recognize the cubes. Furthermore, to increase performance, only the row currently under consideration is converted.

The detection of the cubes works as follows. Using a table, the minimum size of a cube in image pixel is determined. Then, starting from the bottom row and going from left to right in each row, the image is scanned for a continuous chunk of pixels that have the cube color. Image pixels are considered to have the cube color if the deviation from a stored hue value is below a certain threshold. A cube has been found if such a chunk is equal or larger to the initially determined minimum cube size.

Now the robot has to determine the relative location of the cube from its current position, or more correctly, from its position when the image under consideration was taken. The camera of the robots have a fixed inclination. Therefore, the height of the detected object in the image indicates the distance from the robot. The conversion from image row to distance in meter is done using three tables, `row2Meter`, `yFactor` and `yFactorMinus`. All tables have 60 different values, as the images provided by the EyeCam have a resolution of 60 rows * 80 columns. `row2Meter[i]` can be used to get the distance of the cube if it is detected in ith row of the image. The two other tables, `yFactor` and `yFactorMinus`, are used to calculate how far to the left or right the detected object is from the robot. `yFactor` is used for objects left from the camera and `yFactorMinus` for objects right from the camera. Two different tables are used because that way much better results were obtained. However, I suspect with the right calibration, one table should be sufficient. The tables are camera-dependent and have to be created manually for each robot. For each image row the distance at which an object appears on that row is measured. According to [1], an automatic formula based generation of the tables provides unsatisfying results.

The relatively simple detection algorithm is effective. It is fast, easy to implement, and, most importantly, reliable. The measured deviations from the real object distance ranged from 0.5 cm to 5 cm, depending on the distance from the camera. The further away an object is, the more space is covered by a single row in the image.
The disadvantage is, that obviously no other objects with a hue similar to the cube's may be in the area of operation. Furthermore, the manual creation of the tables is time consuming.

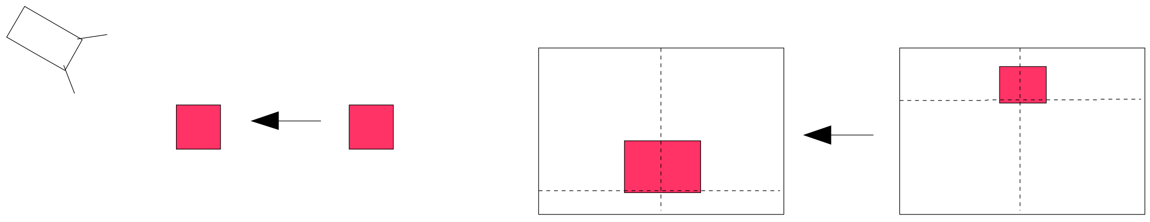


Figure 10. Height in camera image corresponds to distance

6.2.2 Density Estimation

The robot uses its color camera to estimate the cube density. In the implementation without communication the robot must be able to reliably estimate the cube density for a single cluster point to remain in the end. As shown in [7], a single cluster point remains if all robots push cubes to their respectively largest known cluster point. When using radio communication the cube density can be used to determine a common cluster point. As before, all robots push their found cubes to the largest known, common cluster point. Naturally, with radio communication, other values can be used to determine a common cluster point as well - for example the age of the cluster points as in [8]. Then all robots would push newly found cubes to the first discovered cube. One main advantage of a density-based approach is that the value used for the decision is directly related to the objective. So, instead of stringently pushing a number of cubes to the first discovered cube, the reverse procedure is often more advantageous. The disadvantage of the density-based approach is that a density estimation with the camera can not be totally accurate. As a result, and already observed in my experiments, the robots sometimes start pushing cubes back and forth between two cluster points of approximately equal size. An age-based approach would have a well-defined common cluster point. However, through stochastic processes, and as observed in the experiments so far, even in the density-based approach a single cluster point remains in the end.

Another advantage of the density-based approach is that the robots can adjust their cluster point positions. This is necessary as the combination of differential steering system and shaft encoder-based localization is inherently inaccurate [9]. That means after having traveled a long distance, a robot is not able to find exactly back to its old cluster point position. Furthermore, the cluster point position can slowly move due to addition and removal of cubes by other robots.

Based on the density estimation a center of density can be calculated. The position of that center can then be used to adjust a robot's cluster point position regularly.

The density estimation is done by taking three images (one at 0 degrees, one at -20 degrees, one at +20 degrees) and counting the number of pixels with the cube hue. The

density value ranges from 0.0, no pixels with matching hue on any of the images, to 3.0, all pixels in all images have the cube hue. Experiments have shown that the density is highly dependent on the distance from the cubes. Therefore it is necessary to ensure that the density estimation is always done from approximately the same distance.
The center of density is estimated by calculating a weighted mean in the image with the highest density. The position of the center of density is then converted into local robot coordinates as in 6.2.2.

6.2.3 Approaching and Pushing

1. Upon detection of a cube the robot has an estimation of the cube distance
2. The robot moves to a fixed distance to the cube
3. At that defined distance a cube density estimation is performed (Note: if the robot decides to use this location as its new cluster point, the approach is aborted)
4. If the angle between the lines robot-cube and cube-cluster point is below 60 degrees the robot approaches the cube in a straight line and starts pushing
5. Otherwise the robot moves closer to the cube in a straight line until it is about one robot length away
6. Based on the distance information from its infra-red sensors the robot decides whether to drive around the cube from the left or right side
7. The robot drives around the cube until it detects an obstacle in front of it, or until the robot, the cube and the cluster point are aligned
8. If the robot did detect an obstacle and ended up in a bad position to push the cube to the cluster point (the cluster point is behind the robot), it tries to drive around the cube the other way
9. The robot tries to push the cube to its cluster point

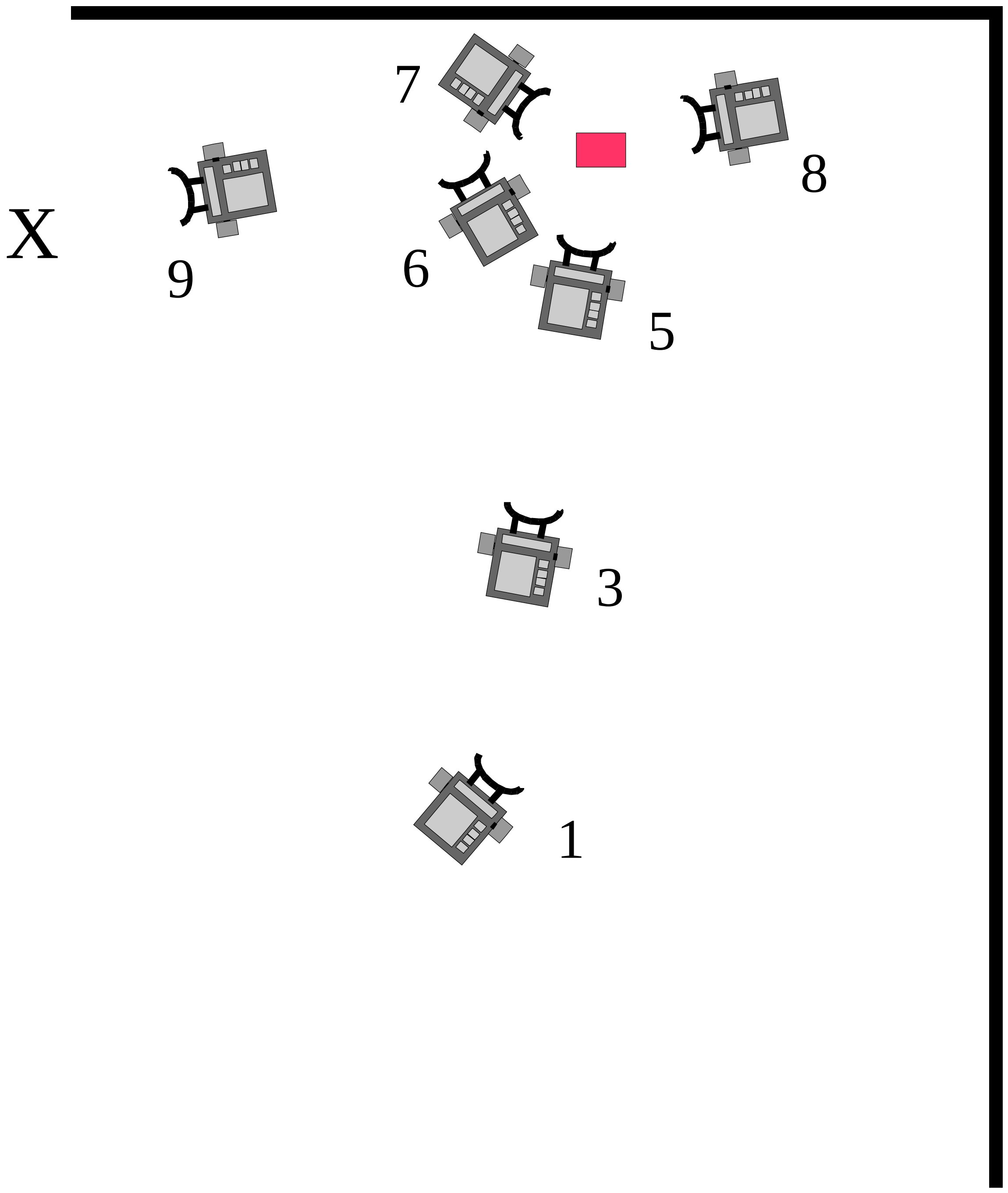


Figure 11. Approaching and pushing a cube

One main advantage of this way of approaching the cube is, that the decision of how to circumnavigate the cube is made at the last moment and based on sensor data. Moreover, this method is reliable, and easy to program and understand. The disadvantage is that this method is a bit slower than for example the approaches used in [1]. But this is not a major limitation as speed has a lower priority here, as opposed to the competitive environments there.

The way of approach ensures in most cases that the robot can push the cube to its cluster point in a direct curve. When pushing the cube, the robot uses its inclined front PSD sensor to determine if it is still in possession of the cube. If the cube is lost, the robots tries to re-detect the cube by backing up and turning 20 degrees to the left and right. If the cube is re-detected the robot continues its push operation, otherwise the robot gives up.

6.2.4 Communication

The robots communicate with each other to speed up the emergence of a common cluster point. Two communicating robots exchange their cluster point positions and densities. They accept the denser of the two clusters as their common one. It is expected that with communication a common cluster point for all robots is found faster.

How can two robots exchange their cluster point positions? The easiest way to exchange two positions can be used if both robots have the same coordinate system. Please note that we are always talking about the global coordinate system of the robot here, as defined in 6.1.1. The local coordinate system of the robot travels with the robot and, for obvious reasons, cannot be used to store its cluster point. They can directly exchange their coordinates. For example:

Robot 1: 'My home is at (2,0)'
Robot 2: 'My home is at (10, 3)'

However, all robots would have to be started from the same point to have the same coordinate system. But then, there would be no need for communication at all. The robots could just push all cubes to their common starting point.

If the robots are not started from the same point and do not have a common coordinate system, they need to (a) synchronize their coordinate systems and (b) then exchange their cluster point positions in the common coordinate system. As explained in section 6.1.1 the robots always have their cluster points at the origin of their coordinate systems. For this reason, step b is not necessary here: once a robot knows the coordinate system of the other robot, it also knows the cluster point of the other robot.

One way to synchronize two coordinate systems is the usage of landmarks. For example:

Person 1: 'My home is 1.2 km south of the lake.'
Person 2: 'My home is 500 m west of the lake.'

If both persons know where the lake is, they can find each other's homes.

However, by using landmarks we decrease the flexibility and robustness of the system. The system could not be deployed at sites without landmarks, and a removal of the landmarks would lead to a collapse of the communication system. Furthermore, the use of landmarks are an introduction of a world model, which contradicts the behavior-based approach.

Landmarks are not necessary when two robots cross each other's paths. If the position of another robot is known exactly, the other robot can send us his position in his coordinate system. By adjusting the received coordinates with his relative position to us, we can calculate our position in his coordinate system. And we have synchronized the two robot's coordinate systems. An example:

Person 1: 'I am 5 km west from my home.'
Person 2: 'I am 3 km south and 200 m north from my home.'

If the two persons can see each other, or now the relative position of each other, they can find each other's home.

But that means the robots have to recognize each other. The EyeBot body has no distinctive hue. Therefore, the camera cannot be used to detect other robots. The other main source of sensor data, the PSDs cannot be used to detect robots either, as they can only return the distances to obstacles.
But there is one way to reliably detect other robots. Only robots collide with each other! A single robot does not collide with walls, the signals delivered by the PSD sensors arrive in time to avoid them. However, another robot moves as well, and can double the relative speed at which the robots approach each other. Furthermore, the robot bodies are uneven, and the PSD sensors do not always detect them in time. The collision and the resulting closeness of the robots is actually helpful in determining the relative

position to each other.

To sum it up: when a robot (called R) collides with an obstacle, it probably hit another robot. R broadcasts a message containing its current position in its own coordinate system and its cluster density. Then R waits for responses. If only a single response is received, the message is probably from the other robot (called S). If no or more than one response is received, the communication procedure is aborted, as R does not know with who it collided. This identification procedure is needed because of the properties of 'EyeNet' radio communication system. It is a virtual token ring with explicit communication through a unique robot id, which means we have to always make sure that we are indeed communicating with the right robot.

If S has been successfully identified, R and S compare their cluster point densities and choose the cluster point with the higher density as their common one. Let us assume that S has the lower cluster density. Then S uses the received coordinates and the known relative position of R to calculate its own position in R's global coordinate system. As R's cluster point is at the origin of its global coordinate system, they have successfully synchronized their coordinate systems, and have a common cluster point now.

6.2.5 Exploration

The Explorer has a simple yet effective behavioral pattern. The exploring behavior is active until a cube is found, the robot is stalled, or another robot is detected.

When meeting an obstacle the robot just turns away from the obstacle until all PSD sensors are free (i.e. the reported distances exceed a certain threshold) again. If the robot is started in a corner, or if the robot is surrounded by obstacles, it can happen that the robot gets trapped and keeps turning, because with no orientation all three PSD sensors are free. Then, the obstacle threshold is slowly decreased so that the robot can find a way out. When all sensors are free again, the threshold is set back to its old value.

The robots try to avoid their own cluster points to prevent them from disarranging their cluster. However, the robots sometimes become stuck between their cluster point and a wall if the cluster point is close to the wall. The sequence is about as follows: The robot tries to avoid its cluster point by turning away. Then it detects the wall and turns away from it, and again the robot faces the cluster point. The cycle starts all over. To prevent this, the robot is allowed to cross its cluster point when it is caught in such a cycle, even at the risk that the cluster is disarranged. Undesirable cycles of such kind are easily detected by tracing sequences where two occurrences of cluster point avoidance are separated by a single occurrence of obstacle avoidance. In the current implementation sequences of c-o-c, c-s-o-c, c-o-s-c, or c-s-o-s-c trigger the allowance to cross the cluster point (c denotes cluster point avoidance, o denotes obstacle avoidance, s denotes straight driving). To simplify the implementation the robot currently does not trace sequences of o-c-o, o-s-c-o, o-c-s-o, and o-s-c-s-o. They indicate the same undesirable cycle, only starting with obstacle avoidance instead of cluster point avoidance. We just accept that one additional redundant turn until the original trigger is activated.

If none of the mentioned cases apply, the robot just drives straight.

6.3 Programming Issues

6.3.1 Singleton Classes

Some classes need to initialize underlying hardware components of the robot, or they need to be accessed by a number of different classes. The initializations of the hardware components are done in the class constructors of *Drive*, *PSDs*, *Camera*, and *Radio*. We have to make sure that only one instance of these classes are created, as multiple initializations and allocations of the hardware components would result in errors. In the case of *SIR*, all classes that need to access it should access the same instance. Otherwise several *SIR* threads would be running in parallel, a massive consumption of processing time.

The singleton design pattern as described in [2] can be used to solve the problem. The intent of the singleton pattern is to "ensure a class has only one instance, and provide a global point of access to it". This is achieved by declaring the constructor as protected. This way, only the class can create instances of itself. Then, the class just needs to create a single instance of itself and provide a public method to access that instance. An example:

```
class Drive
{
  public:
     //Used to access the single instance of this class
     static Drive* getDrive(void);
  protected:
     //The constructor, can only be called by this class
     Drive();
  private:
     //the single instance of this class
     static Drive aDrive;
};
Drive Drive::aDrive;
Drive* Drive::getDrive(void)
{
  return &aDrive;
}
Drive::Drive()
{
  //initializations
  ...
}

void main(void)
{
  Drive* drive = Drive::getDrive();
  //drive is a pointer to the only instance of Drive
  //and can be used to control the robot drive now
}
```

Code 1. Example of singleton class

6.3.2 The Thread Class

Some modifications were necessary to make the *Thread* class work with the RoBIOS library. First, I will quickly show how it was actually supposed to look like. Then I will explain why and what modifications were necessary.

The program consists of up to four threads, as defined in the software architecture. The *Thread* class was designed to (a) make the use of threads easier by encapsulating the complexity in a separate class, and (b) to avoid having to rewrite the same code for all threaded classes, with all disadvantages in modifiability, consistency and comprehensibility.
The following pseudo-code illustrates how it was initially intended.

```
class Thread
{
  public:
     // Constructor for Thread
     Thread (char* name, int stackSize, int priority, int id);

     // Initializes the thread, returns true if successful
     bool spawn ();

  protected:
     //must be overwritten by the inheritor with its own code
     virtual void run(void);
};



//We create a threaded class called 'MyThread'. MyThread
inherits all the properties it needs to be a thread from the
'Thread' base class. We just need to overwrite the 'run' method
with our own code

class MyThread : public Thread
{
  private:
     void run(void);
};
void MyThread::run(void)
{
     //Here we enter the code for MyThread
}

void main(void)
{
     //create an instance of MyThread
     MyThread myThread("thread1", 8192, MAX_PRI, 1);

     //initialize the instance
     myThread.spawn();

     //now the instance can be used

     ...

     return 0;
}
```

Code 2. How to create threads (as initially intended)

The example above shows that all the complexity in connection with multi-threading is encapsulated in the *Thread* base class alone. Any class that needs to be a thread can just inherit from *Thread* and overwrite the virtual `run` method with its own code. The *Thread* class offers a small set of simple methods (`kill`, `sleep`, ...) to control the thread.

Unfortunately, some properties of the multi-tasking functions provided by the RoBIOS library made a few modifications necessary.

1. When initializing a thread, a pointer to a C-function containing the code to be executed is expected.
2. All threads must be initialized (`spawn`ed) in `main`.

1. I will call a pointer to a standard C function *pointer-to-function* and a pointer to a C++ method *pointer-to-member-function.* There is a major difference between these two. A method does not make sense without its associated instance. The type of a pointer-to-function is different from a pointer-to-member-function. Let us consider the function `int sum (int a, int b)` and the method `int Algebra::sum(int a, int b)`. The type of the function is `int (*) (int, int)`, whereas the type of the method is `int (Algebra::*) (int, int)`. Fortunately, there is an easy way out. Pointer-to-**static**-member-functions are compatible with regular pointer-to-functions [4].

That means if we declare the `run` method of a thread as static we can pass it to the RoBIOS library. The problem is that static methods can only use other static methods and variables. The reasons is that a static method, which is shared by all instances, would not know which non-static method or variable to use if there were multiple instances of the same class. Thus, the static property would have to be applied to all used (sub-) methods and (sub-) variables, and propagate through the whole class structure. That is why the threads' `run` method are left as non-static and static wrapper methods called `staticRun` are used. A pointer to the wrapper method is passed to `OSSpawn`, it does nothing but to call `run`. Attentive readers will have realized by now that there is a catch to this solution. If static methods cannot call non-static ones, how can `staticRun` call `run`? As aforementioned the reason why static method cannot use non-static members is because it is not clear which non-static member to use if there are multiple instances of the same class. That means `staticRun` needs a pointer to the instance whose `run` method is to be called. This pointer is stored in a static member variable called `me`.

According to C++ convention, methods cannot be static and virtual at once. In the initial design `run` was declared as virtual which results in dynamic binding. That means the program determines at run-time which `run` method to use, the one of `Thread` or the one of `MyThread`. Without the virtual keyword the compiler determines at compile-time which `run` method is used, the one of `Thread`. To ensure that MyThread indeed uses its own `run` method, it is necessary to overwrite the thread initialization method (`spawn`) as well. Let us have a look at the final solution.

```
class Thread
{
  public:
     // Constructor for Thread
     Thread (char* name, int stackSize, int priority, int id);

     // Initializes the thread, returns true if successful
     bool spawn ();

  protected:
     //must be overwritten by the inheritor with its own code
     void run(void);
};

class MyThread : public Thread
{
  public:
     //without virtual run methods every thread needs
     //its own spawn
     bool spawn(void);

  private:
     void run(void);

     //static wrapper method for run
     static void staticRun(void);

     //stores a pointer to an instance of MyThread for staticRun
     //initialized in the constructor
     static MyThread* me;
};
MyThread::MyThread()
{
     me = this;
}
void MyThread::run(void)
{
     //Here we enter the code for MyThread
}
void MyThread::staticRun(void)
{
     me->run();
}

void main(void)
{
     //looks exactly the same as in the easy example

     //create an instance of MyThread
     MyThread myThread("thread1", 8192, MAX_PRI, 1);

     //initialize the instance
     myThread.spawn();

     //now the instance can be used

     ...

     return 0;
}
```

Code 3. How to create threads in the program

As seen in the example, this final solution is as easy to use as the initial proposal, though the external framework is a little bit trickier.

2. As all thread must be initialized (`spawn`ed) in main, the threads must be known in main. This is partly solved by creating global threads, as in the case of *Commander* and *Broadcaster*, by using singletons, as in the case of *SIR*, or, whenever possible, by using local instances, as in the case of *UI*. Why three different approaches? Obviously, using a local instance is the best solution. But that only works if just main needs access to the thread, as with *UI*. *SIR* was designed as singleton in the first place, as a number of classes need to access it. Therefore, `main` can use the built-in mechanism to access the instance. Finally, none of the mentioned cases apply to *Commander* and *Broadcaster*, so we use global instances.

6.3.3 Race Conditions

Like in most multi-threaded programs, we have to deal with inter-thread-communication and synchronization. In the current implementation, thread interaction only takes place between SIR and the respective active behavior. Through a reasoned implementation of the methods and variables, `getCubePosition`, used by Pusher, is the only method where a race condition can possibly occur. The sequence: Pusher reads the x coordinate of a cube position. Then a context-switch takes place, and SIR updates the cube position. Another context switch, and Pusher reads the y coordinate of the new cube position. However, this race condition is not critical at all. SIR just keeps providing better estimations of the cube position, and steadily approaches the correct values with increasingly smaller changes. Therefore the coordinates do not need to be protected.

6.3.4 Cyclic Dependencies

There is a cyclic dependency between the classes *Behavior*, *Commander* and *Explorer*/*Pusher*/*Avoider*/*Communicator*, as we can see in the system architecture. This cycle arises as

(a) the *Behavior* base class provides access to the commander; thus a behavior can tell the commander that it wants to cede control or wants to sleep for some time;

(b) all behaviors are derived from the *Behavior* base class;

(c) the commander knows each behavior in order to control it.

The compiler is not able to solve such a dependency on its own. One of the classes needs to be compiled first, but each of them is dependent on the other two and, through the coupling, on itself. We have to break the cycle. In this case it can be done using a forward declaration and a special include pattern [3].

The *Behavior* class does not actually need to know the details of the *Commander* class, as it only stores a pointer to one. Pointers are the same, no matter what they point to, therefore we do not need the definition of the class in order to store the pointer. That means no `#include "Commander.h"` command is needed in *Behavior.h*. However, simply taking it out will result in an error during compilation. So we need to let the compiler know that there is a *Commander* class. This is done with by replacing the include line by a so-called forward declaration - a class definition without a body. This allows us to break the dependency between *Behavior.h* and *Commander.h.*

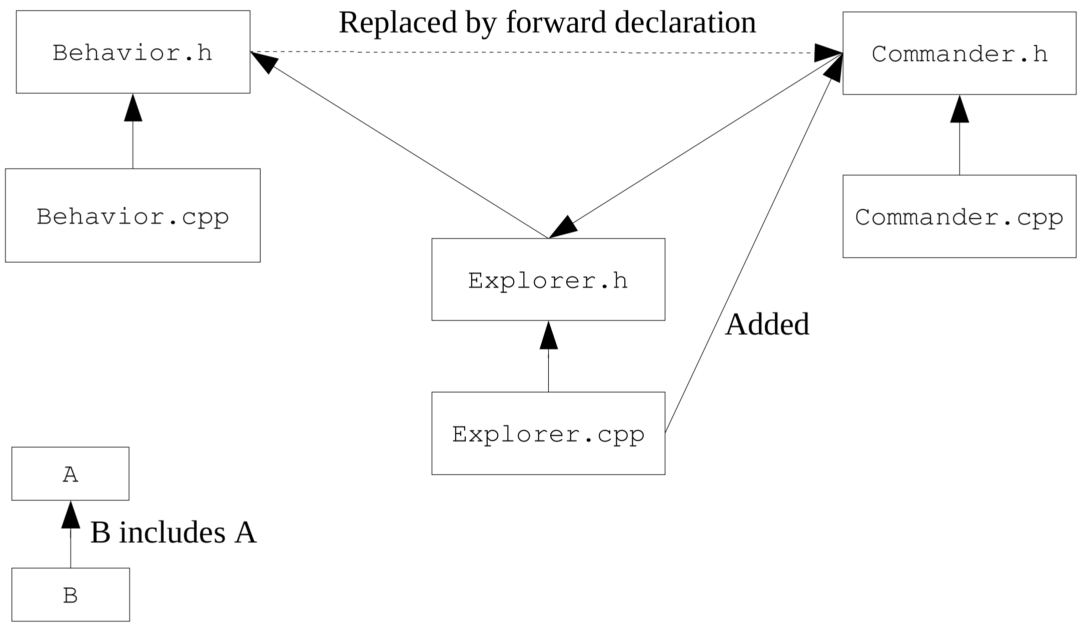


Figure 12. Dissolving the cyclic dependency

Of course, the implemented behaviors (*Explorer*, *Avoider*...) do need to know the definition of the *Commander* class. That is why *Commander.h* needs to be included by them.

6.4 From Real Robots to the Simulator

Generally, most parameters had to be adapted, as the simulator uses different from real robots, different camera pos, angle, balls instead of cubes.

When pushing a cube, the robot uses its inclined front PSD sensor to determine if it still possesses the cube. The PSD sensors of the simulated robots cannot be inclined. Therefore, in the simulation the robots cannot detect the loss of a cube. They keep driving back to their cluster points even if they have lost their cubes.

In the simulator RoBIOS functions may not be called in the class constructors. Therefore I had to introduce public initialization methods that must be called before a class is used.

6.4.1 Multiple Robots on EyeSim

The EyeSim simulator is not able to simulate multiple robots with a program that uses global variables. All robots would share the same global variables [6]. The same holds for static variables. So a few modifications are required when porting the program to EyeSim. Fortunately, because of the use of C++, no global variables are needed for the interprocess communication. An analysis showed that the program contains two critical global variables and nine critical static variables. The two global variables, one instance of *Commander* and one instance of *Broadcaster,* are necessary because these two threads must be initialized in `main`, but they are used in a different part of the program. The nine static variables are the five instances of the singleton classes (*Drive*, *Camera*, *PSDs*, *Radio* and *SIR*) and the four threads (*Broadcaster*, *Commander*, *UI* and, again, *SIR*). More static variables are used in the program as messages for the interprocess communication, but those are constant and can be shared.

The solution is quite straightforward. An array is created for each global/static variable that cannot be shared. The array size corresponds to the number of robots to be used in the simulation. As each robot has a unique id, accessible by `OSMachineID`, every robot has its own instances of the critical variables. The robot id corresponds to the array index. If the array size is set to 1, the resulting program is virtually equal to the original one for real robots.

```
//Example for global variables:

//creation of a global array of commanders
Commander g_commander[NUMBER_OF_ROBOTS];

//later each robot initializes and uses its own commander
g_commander[OSMachineID()].initialize();


//Example for static variables:

//creation of a static array of drives
Drive Drive::drives[NUMBER_OF_ROBOTS];

//later each robot gets and uses only its own instance
return &drives[OSMachineID()];
```

Code 4. Arrays for all global/static variables in EyeSim

6.4.2 Stalled Drive Information

Another difference between real robots and simulated ones is the reliability of the information whether the wheels are stalled. Evidently, in the simulator this information is reliable and instantaneous. When using real robots, this information is not always reliable. When the robot accelerates it has to overcome a certain frictional force before it starts moving. Depending on the parameters of the drive controller it can take some time until the robot starts moving. This is often wrongly interpreted as a stalled wheel. Therefore, when using real robots, a counter is incremented each time the robot receives the information that one of its wheels is stalled. The robot decides that its wheels are indeed stalled only if the counter exceeds a certain limit.

7. Testing

Real-environment testing was performed at the Mobile Robot Lab using EyeBots, cubes and a plane as specified in the system requirements specification.

The tests covered only the main functionality. Tests of the overall system, sub-system tests on behavior level (i.e. compliance of the behaviors with the system model), and tests of critical components (cube detection and density estimation) were performed. A major advantage of the behavior-based approach is that the behaviors can be implemented and tested independently from each other. Generally, classes of the robot package were not tested as their main task is just to pass commands to the underlying robot library functions. Errors in these classes were quickly revealed when testing components of the clustering package. Moreover, the robots are designed for robustness, which means small deviations (e.g. of the sensor readings) should have no major impact on the overall behavior.

All tests were passed to our satisfaction. See the test protocol for details.

8. Evaluation and Future Work

The implemented cube detection algorithm is reliable and fast. It seems to perform better than the original algorithm of Birgit Graf, on which my implementation is based. I assume one reason is that I specified the hue to be detected more accurately. When experimenting with Birgit Graf's program I realized that the hue she specified was about 10 hues off the optimal one. Secondly, I modified the conversion from RGB to HSV such that the result 'no hue' occurs considerably less often.

Worth improving is the calculation of the distances of detected cubes from the robot. At present, three tables are used for this conversion. However, with the right calibration, two tables should be sufficient.

In a few cases a robot has collided with a wall because it thought there was enough space. As aforementioned the PSD readings are too high when a robot is close to a wall. However, this problem is not critical as the robot quickly frees itself.

In the current implementation the robots have trouble to cluster cubes that are close to walls. It might be useful to re-design the approaching and pushing behavior such that cubes close to walls can also be collected.

Due to time constraints the communication by broadcasting was not implemented. As argued before, with a global starting point, the robots could just push all cubes to that location. There would be no need for communication at all. Communication by broadcasting was intended as an evolutionary step towards the final solution, local communication. However, I implemented the one-on-one communication directly.

Tests were performed to verify the compliance of the system behavior with the system model. The clustering terminated successfully in most cases, and scalability and robustness of the system could be demonstrated. However, more advanced tests could be performed, for example to compare the efficiency of swarm clustering systems and single-robot systems, to determine the optimal robot density for a given area, and so on.

Currently, the robots can only cluster cubes on empty planes. The robots assume that they do not need to avoid walls, only other robots. Once a robot has a cube, it heads for its cluster point in a direct curve. If the robot hit a wall, it would lose the cube, and the robot would not be able to retrieve that cube. The program could be extended such that the robots would able to cluster cubes in more complex operational areas. For example the floor of an office building could be recreated. That would be an excellent example of how multi-robot systems could be used for garbage collection.

9. Conclusion

In this thesis I presented the swarm clustering algorithm I developed. The algorithm was implemented for both real robots and a simulator. Software engineering principles were observed throughout the project.

For the algorithm I designed a versatile framework for behavior-based control. I implemented four behaviors: exploration, avoidance, pushing and communication. The robots use only local sensors and on-board image processing. A reliable image recognition method (for the cubes to be clustered) was implemented. Furthermore, an inter-robot communication system was incorporated. It is used to speed up the determination of a common cluster point by synchronizing the coordinate systems, and thus the cluster points, of robots that meet each other.

First experiments showed the robustness and flexibility of the developed swarm clustering algorithm. The robots could be started from any position in the operational area. And robots could be added or removed during operation. The algorithm still terminated successfully.

To the best of my knowledge, this is the first image processing-based collective clustering algorithm with communication.

The results of this project will be presented at the AMiRE 2003 (Autonomous Mini-robots in Research and Edutainment) conference in Brisbane, Australia.

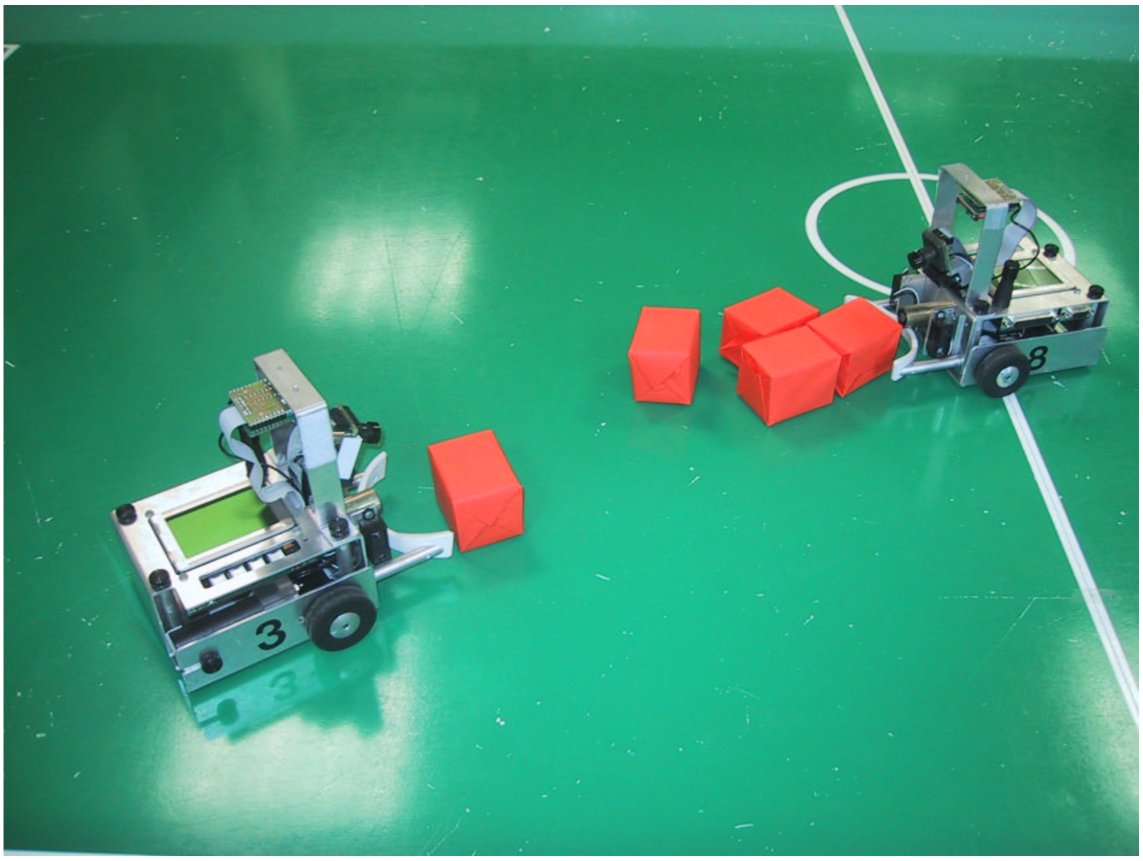


Figure 13. Clustering with EyeBots

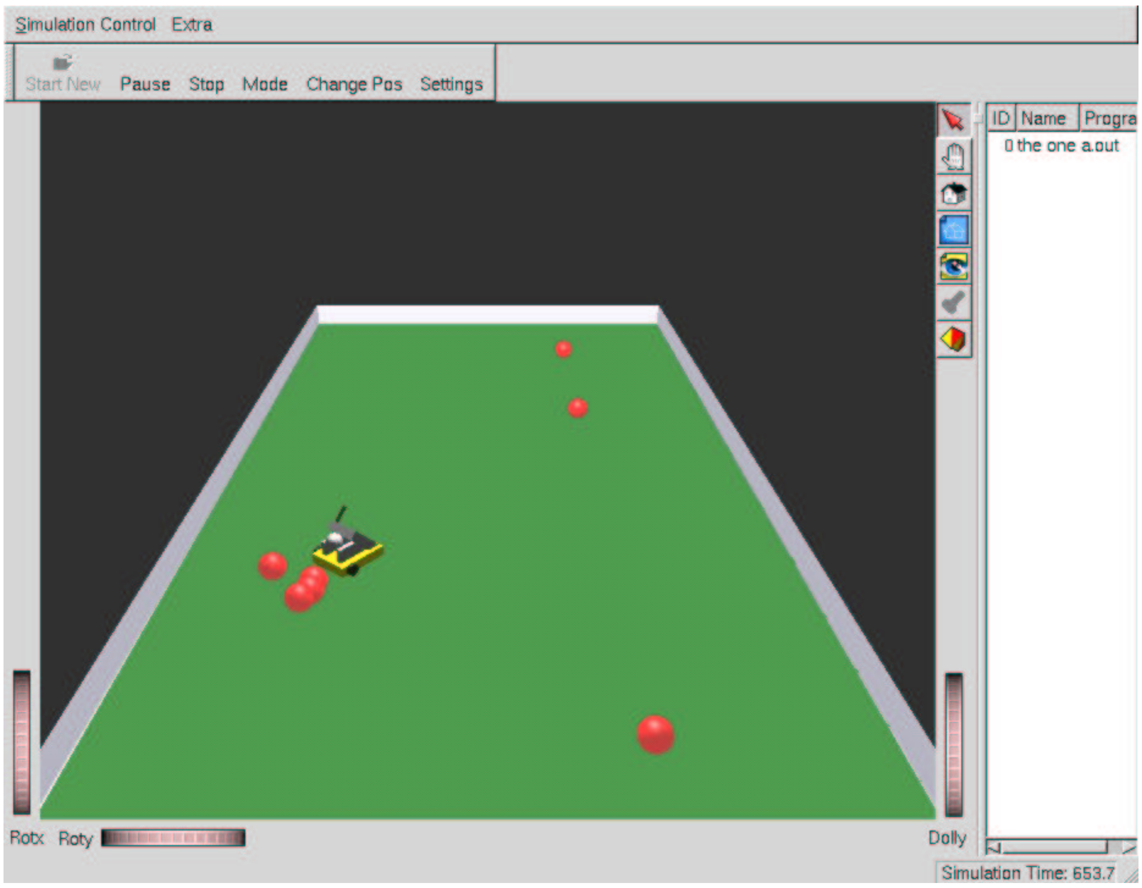


Figure 14. Clustering on the EyeSim simulator

References

[1] **Graf, Birgit; Bräunl, Thomas:** *Robot Soccer*, Diploma Thesis No. 1700, Faculty of Computer Science, University of Stuttgart, Germany, 1999

[2] **Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John:** *Design Patterns*, Addison-Wesley, 1995

[3] **Sizer, Ben:** *Organizing Code Files in C and C++*, Available from http://www.gamedev.net/reference/programming/features/orgfiles/page3.asp

[4] **Cline, Marshall:** *C++ FAQ Lite*, Available from http://www.parashift.com/c++-faq-lite/pointers-to-members.html

[5] **Gerardin, Peggy:** *Content-based Image Retrieval*, Audio Visual Communications Laboratory, Ecole Polytechnique Fédérale de Lausanne, Switzerland, Available from http://lcavwww.epfl.ch/Teaching/PHOTO/Projects_2002/Proj3.html

[6] Waggershauser, Axel; Venkitachalam, Daniel; Bräunl, Thomas: *EyeSim 5*, Available from http://robotics.ee.uwa.edu.au/eyebot/index.html

[7] **Martinoli, A.; Goodman, R.; Holland, O.:** *Swarm Intelli*gence, Collective Robotics Research Group, California Institute of Technology, Pasadena, California, USA, 2001, Available from **http://www.coro.caltech.edu/Courses/EE141/course.html**

[8] **Iske, B.; Rückert, U.:** *Cooperative Cube Clusterung using Local Communication*, Autonomous Robots for Research and Edutainment - AMiRE 2001, Proceedings of the 5th International Heinz Nixdorf Symposium, Paderborn, Germany, 22.-24. Oct., 2001, pp. 333-334.

[9] **Zelinsky, Alex:** *Mobile Robotics*, ENG 4527, Department of Systems Engineering, Research School of Information Sciences and Engineering, Australian National University, Canberra, Australia, Available from http://www.syseng.anu.edu.au/~alex/IntroW1A.pdf

[10] **Waggershauser, Axel:** *Simulating Small Mobile Robots*, Department of Electrical Engineering and Information Technology, University of Kaiserslautern, Germany, 2002, Available from http://robotics.ee.uwa.edu.au/eyebot/index.html

[11] **Göhner, Peter:** *Software Engineering for Real-Time Systems*, Institute of Industrial Automation and Software Engineering, University of Stuttgart, Germany, Available from http://www.ias.uni=stuttgart.de/vorlesungen/ser/index.html

[12] **Storey, Daniel:** *Wireless Communication for Intelligent Robotic Agents*, Department of Electrical and Electronic Engineering, University of Western

Australia, Perth, 1998, Available from http://robotics.ee.uwa.edu.au/eyesoccer/papers

[13] **Marsland, Stephen:** *Autonomous Mobile Robots*, Department of Computer Science, University of Manchester, UK, Available from http://www.cs.man.ac.uk/~marslans/CS3451

[14] **Beckers, R.; Holland, O.; Deneubourg, J-L.:** *From local actions to global tasks: stigmergy in collective robotics,* Artificial Life 4, eds. R Brooks and P Maes, MIT Press, 1994

Figure Sources

Figure 1. The EyeBot

Photo of EyeBot from Christoph Braunschädel, Department of Electrical and Electronic Engineering, University of Western Australia, Perth, 2002

Figure 3. Schematic illustration of a model-based robot control system

Reproduced from [9]

Figure 4. Schematic illustration of a behavior-based robot

Reproduced from [9]

Figure 5. Multi-robot systems in research

Reproduced from [7]

Figure 6. Types of collective robotics

Reproduced from [7]

Figure 7. The HSV cone

Source: Computer Science Educational Lab, University of Colorado at Boulder, USA, Available from http://www-ugrad.cs.colorado.edu/~csci4576/Figures/hsv.gif

Figure 8. The RGB cube

Source: CVonline: The Evolving, Distributed, Non-Proprietary, On-Line Compendium of Computer Vision, School of Informatics, University of Edinburgh, Available from http://www.dai.ed.ac.uk/CVonline/LOCAL_COPIES/OWENS/LECT14/rgb.gif

Figure 10. Height in camera image corresponds to distance

Reproduced from [1]

Appendix A – Used Software

The software was developed on a Linux-based PC system using the modified GNU C/C++ compiler for EyeBots. The system model, the class diagrams, and the code stubs were created with Rational Rose. Doxygen was used for code documentation. All documents were created with OpenOffice. For details on the used software please see the system requirements specification.

Declaration

I hereby declare that this submission is my own work and that I only used the referenced aids.

Perth,

Jia L. Du